Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Contents

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Authors retain copyright to all submitted papers, but have granted unlimited redistribution rights to all as a condition of submission.

Enabling Docking Station Support for the Linux Kernel

Is Harder Than You Would Think

Kristen Carlson Accardi
Open Source Technology Center, Intel Corporation
kristen.c.accardi@intel.com

Abstract

Full docking station support has been a feature long absent from the Linux kernel—for good reason. From ACPI to PCI, full docking station support required modifications to multiple subsystems in the kernel, building on code that was designed for server hot-plug features rather than laptops with docking stations. This paper will present an overview of the work we have done to implement docking station support in the kernel as well as a summary of the technical challenges faced along the way.

We will first define what it means to dock and undock. Then, we will discuss a few variations of docking station implementations, both from a hardware and firmware perspective. Finally, we will delve into the guts of the software implementation in Linux—and show how adding docking station support is really harder than you would think.

1 Introduction and Motivation

It's no secret that people have not been clamoring for docking station support. Most people do not consider docking stations essential, and indeed some feel they are completely unnecessary. However, as laptops become thinner and lighter, more vendors are seeking to replace functionality that used to be built into the laptop with a docking station. Commonly, docking stations will provide additional USB ports, PCI slots, and sometimes extra features like built in media card readers or Ethernet ports. Most vendors seem to be marketing them as space saving devices, and as an improved user experience for mobile users who do not wish to manage a lot of peripheral devices.

We embarked on the docking station project for a few reasons. Firstly, we knew there were a few members of the Linux community out there who actually did use docking stations. These people would hopefully post to the hotplug PCI mailing lists every once in a while, wondering if some round of I/O Hotplug patches would enable hot docking to work. Secondly, there was a need to be able to implement a couple scenarios that dock stations provide a convenient test case for. Many dock stations are actually Peer-to-peer bridges with a set of buses and devices located behind the bridge. Hot add with devices with P2P bridges on them had always been hard for us to test correctly due to lack of devices. Also, the ACPI _EJD method is commonly used in AML code for docking stations, but this method can also be applied to any hot-pluggable tree of devices. Finally, we felt that with the expanding product offerings for dock stations, filling this feature gap would eventually become important.

2 Docking Basics

There are three types of docking that are defined.

- **Cold Docking/Undocking** Laptop is booted attached to the dock station. Laptop is powered off prior to removal from the dock station. This has always been supported by Linux. The devices on the dock station are enumerated as if they are part of the laptop.
- **Warm Docking/Undocking** Laptop is booted either docked or undocked. System is placed into a suspend state, and then either docked or undocked. This may be supported by Linux, assuming that your laptop actually suspends. It depends really on whether a driver's resume routine will rescan for new devices or not.
- **Hot Docking/Undocking** Laptop is booted outside the dock station. Laptop is then inserted into the dock station while completely powered and operating. This has recently had limited support, but only with a platform specific ACPI driver. Hotplugging new devices on the dock station has never been supported.

Docking is controlled by ACPI. ACPI defines a dock as an object containing a method called _DCK. An example dock device definition is shown in Figure 1.

_DCK is what ACPI calls a "control method". Not only does it tell the OS that this ACPI object is a dock, it also is used to control the isolation logic on the dock connector.

```
Device (DOCK1) {
        Name(_ADR, ...)
        Method(_EJ0, 0) {...}
        Method(_DCK, 1) {...}
}
```

Figure 1: Example DSDT that defines a Dock Device

When the user places their system into the docking station, the OS will be notified with an interrupt, and the platform will send a Device Check notify. The notify will be sent to a notify handler and then that handler is responsible for calling the _DCK control method with the proper arguments to engage the dock connector.

_DCK as defined in the ACPI specification is shown in Figure 2. Assuming that _DCK returned successfully, the OS must now re-enumerate all enumerable buses (PCI) and also all the other devices that may not be on enumerable buses that are on the dock.

Undocking is just like docking, only in reverse. When the user hits the release button on the docking station, the OS is notified with an eject request. The notify handler must first execute _DCK(0) to release the docking connector, and then should execute the _EJ0 method after removing all the devices that are on the docking station from the OS.

The _DCK method is not only responsible for engaging the dock connector, it seems to also be a convenient place for system manufacturers to do device initialization. This is all implementation dependent. I have seen _DCK methods that do things such as programming a USB host controller to detect the USB hub on the dock station, issuing resets for PCI devices, and even attempting to modify PCI config space to assign new bus numbers[1] to the dock bridge.

[1] Highly unacceptable behavior

This control method is located in the device object that represents the docking station (that is, the device object with all the _EJx control methods for the docking station). The presence of _DCK indicates to the OS that the device is really a docking station.

_DCK also controls the isolation logic on the docking connector. This allows an OS to prepare for docking before the bus is activated and devices appear on the bus [1].

Arguments:

Arg0

1 Dock (that is, remove isolation from connector)

0 Undock (isolate from connector)

Return Code:

1 if successful, 0 if failed.

Figure 2: _DCK method as defined in the ACPI Specification

The only way to know for sure what the _DCK method does is to disassemble the DSDT.

3 Driver Design Considerations

There are platform specific drivers in the ACPI tree. The `ibm_acpi` drier had previously implemented a limited type of docking station support that would only work on certain ibm laptops. Essentially, this driver would hard code the device name of the dock to find the dock, and then would execute the _DCK method without rescanning any of the buses or inserting any of the non-enumerable devices. It suffers from being platform specific, which is not ideal. We wanted to make a generic solution that would work for most platforms.

We originally assumed that all dock stations were the same: a dock bridge would be located on the dock station, which was a P2P bridge, and all devices would be located behind the P2P bridge. The IBM ThinkPad Dock II is an example of this type of implementation, shown in Figure 4. The same driver (`acpiphp`) that could hotplug any device that had a P2P bridge on it could be used to hotplug the dock station devices, with the minor addition of needing to execute the _DCK method prior to scanning for new devices.

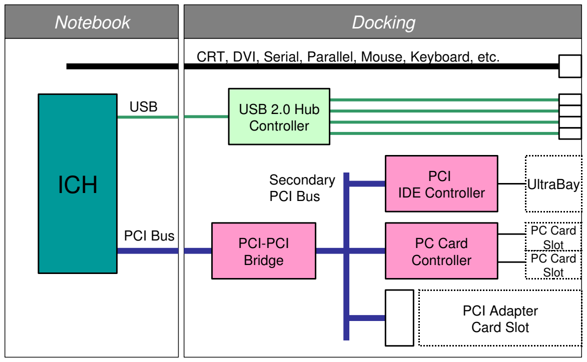


Figure 4: The IBM ThinkPad Dock II ©2006, Noritoshi Yoshiyama, Lenovo Japan, Ltd—Used by Permission

These were bad assumptions.

3.1 Variations in dock device definitions

The dock device definition for a few IBM ThinkPads that I had available is shown in Figure 5. The physical device is a P2P bridge.

```
IBM_HANDLE(dock, root, "\\_SB.GDCK",     /* X30, X31, X40 */
           "\\_SB.PCI0.DOCK",    /* 600e/x,770e,...,X20-21 */
           "\\_SB.PCI0.PCI1.DOCK",       /* all others */
           "\\_SB.PCI.ISA.SLCE",         /* 570 */
    );
```

Figure 3: Defining a dock station in ibm_acpi.c

It appears to fit the ACPI definition of a standard PCI hotplug slot, in that it exists under the scope of PCI0, it has an _ADR function, and it is ejectable (has an _EJ0). It contains the _DCK method, indication that it is a docking station as well. This was our original view of the docking station.

```
T20,T30,T41, T42 look like this:
Device (PCI0)
   Device (DOCK)
   {
      Name (_ADR, 0x00040000)
      Method (_BDN, 0, NotSerialized)
      Name (_PRT, Package (0x06)
      Method (_STA, 0, NotSerialized)
      Method (_DCK, 1, NotSerialized)
      Method (_EJ0, ...)
```

Figure 5: IBM T20, T30, T41, T42 DSDT

Unfortunately for us, not all dock stations are the same. Sometimes system manufactures create a "virtual" device to represent the dock. It simply calls methods under the "real" dock bridge. In this case, the `acpiphp` driver will not recognize the GDCK device as an ejectable slot because it has no _ADR. In addition, it will not recognize the "real" dock device as an ejectable PCI slot because _EJ0 is not defined under the scope of the Dock(), but instead under the virtual device GDCK. An example of this type of DSDT is shown in Figure 6. There are also dock stations that do not

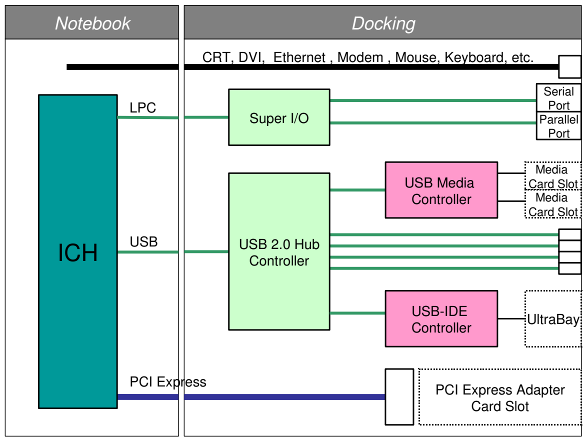


Figure 7: The Lenovo ThinkPad Advanced Dock Station
©2006, Noritoshi Yoshiyama, Lenovo Japan, Ltd—Used by Permission

utilize a P2P bridge for PCI devices, such as the Lenovo ThinkPad Advanced Dock Station, shown in Figure 7. In addition, there are dock stations that do not have any PCI devices on them at all. This made using the ACPI PCI hotplug driver a bit nonsensical. However, the normal ACPI driver model also didn't work, because ACPI drivers will only load if a device exists. However, we decided to move the implementation from the PCI hotplug driver into ACPI, because there really was nowhere else to put it.

In order to decouple the dock functionality from the hotplug functionality, the dock driver needs to allow other drivers to be notified upon a dock event, and also to register individual hotplug notification routines. This way, the dock

```
Scope (_SB)
    Device(GDCK)
        Method (_DCK, 1, NotSerialized)
        {
            Store (0x00, Local0)
            If (LEqual (GGID (), 0x03))
            {
                Store (\_SB.PCI0.LPC.EC.SDCK (Arg0), Local0)
            }
            If (LEqual (GGID (), 0x00))
            {
                Store (\_SB.PCI0.PCI1.DOCK.DDCK (Arg0), Local0)
            }
            Return (Local0)
        }
        Method (_EJ0, 1, NotSerialized)
            ...
    Device (PCI1)
        Device (DOCK)
        {
            Name (_ADR, 0x00030000)
            Name (_S3D, 0x02)
            Name (_PRT, Package (0x06)
```

Figure 6: Alternative dock definition

driver can just handle the dock notifications from ACPI, and individual subsystems/drivers can handle how to hotplug new devices. In the case of PCI, `acpiphp` can still handle the device insertion, but it will not be used if there are no PCI devices on the dock station.

4 Driver Implementation Details

The driver is located in `drivers/acpi/dock.c`. It makes a few external functions available to drivers who are interested in dock events.

4.1 External Functions

`int is_dock_device(acpi_handle handle)`
This function will check to see if an ACPI device referenced by handle is a dock device. This means that the device either is a dock station, or a device on the dock station.

`int register_dock_notifier(struct notifier_block *nb)`
Sign up for dock notifications. If a driver is interested in being notified when a dock event occurs, it can send in a `notifier_block` and be called right after _DCK has been executed, but before any devices have been hotplugged.

`int unregister_dock_notifier(struct notifier_block *nb)`
: Remove a driver's `notifier_block`.

`acpi_status register_hotplug_dock_device (acpi_handle, acpi_notify_handler, void *)`
: Pass an ACPI notify handler to the dock driver, to be called when a dock event has occurred. This allows drivers such as `acpiphp` which need to re-enumerate buses after a dock event to register their own routine to handle this activity.

`acpi_status unregister_hotplug_dock_device(acpi_handle handle)`
: Remove a notify handler from the dock station's hotplug list.

4.2 Driver Init

At init time, the dock driver walks the ACPI namespace, looking for devices which have defined a _DCK method.

```
/* look for a dock station */
acpi_walk_namespace(
ACPI_TYPE_DEVICE,
ACPI_ROOT_OBJECT, ACPI_UINT32_MAX,
find_dock, &num, NULL);
```

If we find a dock station, then we create a private data structure to hold a list of devices dependent on the dock station, and also hotplug notify blocks.

We can detect devices dependent on the dock by walking the namespace looking for _EJD methods. _EJD is another method defined by ACPI, that is associated with devices that have a dependency on other devices. From the spec:

> This object is used to specify the name of a device on which the device, under which this object is declared, is dependent. This object is primarily used to support docking stations. Before the device indicated by _EJD is ejected, OSPM will prepare the dependent device (in other words, the device under which this object is declared) for removal [1].

So, to translate, all devices that are behind a dock bridge should have an _EJD method defined in them that names the dock.

Drivers or subsystems can register for dock notifications if they control a device dependent on the dock station. Drivers use the `is_dock_device()` function to determine if they are a device on a dock station. This allows for re-enumeration of the subsystem after a dock event if it is necessary. In the case of PCI devices, the `acpiphp` driver is modified to detect not only ejectable PCI slots, but also PCI dock bridges or hotpluggable PCI devices. If it does find one of these devices, then it will request that the dock driver notify `acpiphp` whenever a dock event occurs. When a system docks, the `acpiphp` driver will treat the event like any other PCI hotplug event, and rescan the appropriate bus to see if new devices have been added.

4.3 Dock Events

At driver init time, the dock driver registers an ACPI event handler with the ACPI subsystem. When a dock event occurs, the dock driver event handler will be called. A dock is a `ACPI_NOTIFY_BUS_CHECK` event type. First, the event handler will make sure that we are not already in the middle of docking. This check is needed, because I found on some dock stations/laptop combos that false dock events were being generated by the system—probably due to a faulty physical connection. We ignore these false events. It is also necessary to ensure that the dock station is actually present before performing the _DCK operation. This is accomplished by the `dock_present()` function. `dock_present()` just executes the ACPI _STA method. _STA will report whether or not the device is present.

```
if (!dock_in_progress(ds) &&
dock_present(ds)) {
```

`begin_dock()` just sets some state bits to indicate that we are now in the middle of handling a dock event.

```
begin_dock(ds);
```

dock() will execute the _DCK method with the proper arguments.

```
dock(ds);
```

We confirm that the device is still present and functioning after the _DCK method.

```
if (!dock_present(ds)) {
    printk(KERN_ERR PREFIX
"Unable to dock!\n");
        break;
}
```

We notify all drivers who have registered with the `register_dock_notifier()` function. This allows drivers to do anything that they want prior to handling a hotplug notification. This can be important if _DCK does something that needs to be undone. For example, on the IBM T41, the _DCK method will clear the secondary bus number for the parent of the dock bridge[2]. This makes it a bit hard for `acpiphp` to scan buses looking for new devices. `acpiphp` can register a function that is called by the `notifier_call_chain` that will clean up this mistake prior to calling the hotplug notification function.

[2]also highly unacceptable

```
notifier_call_chain(
&dock_notifier_list, event,
NULL);
```

Drivers or subsystems that need to be notified so that devices can be hotplugged can register a hotplug notification function with the dock driver by using the `register_hotplug_dock_device()` function. `hotplug_devices()` just walks the list of hotplug notification routines and calls each one of them in the order that it was received.

```
hotplug_devices(ds, event);
```

We clear the dock state bits to indicate that we are finished docking.

```
complete_dock(ds);
```

Now we alert userspace that a dock event has occurred. This event should be sent to the acpid program. If a userspace program is ever written or modified to care about dock events, they can use acpid to get those events.

```
if (acpi_bus_get_device(
ds->handle, &device))
    acpi_bus_generate_event(
device, event, 0);
```

Undocking is mostly just the reverse of docking. An undock is a `ACPI_NOTIFY_EJECT_REQUEST` type. Once again, we must not be in the middle of handling a dock event, and the dock device must be present in order to handle the eject request properly.

```
if (!dock_in_progress(ds) &&
dock_present(ds)) {
```

Because undocking may remove the `acpi_device` structure that we need to send dock events to userspace, we send our undock notification to the acpid prior to actually executing _DCK.

```
if (acpi_bus_get_device(
ds→handle, &device))
    acpi_bus_generate_event(
device, event, 0);
```

We also must call all the hotplug routines to notify them of the eject request. This is important to do prior to executing _DCK, since _DCK will release the physical connection and may make it impossible for clean removal of some devices. Finally, we can call `undock()`, which simply executes the _DCK method with the proper arguments.

```
hotplug_devices(ds, event);
undock(ds);
```

The ACPI spec requires that all dock stations (i.e. objects which define _DCK) also define an _EJ0 routine. This must be called after _DCK in order to properly undock. What this routine actually does is system dependent.

```
eject_dock(ds);
```

At this point, a call to _STA should indicate that the dock device is not present.

```
if (dock_present(ds))
    printk(KERN_ERR PREFIX
"Unable to undock!\n");
```

The design of the driver was intentionally kept strictly to handling dock events. For this reason, this is the only thing of interest that this driver does.

5 Conclusions

Dock stations make excellent test cases for hotplug related kernel code. Attempting to hotplug a device which can be a PCI bridge with a tree of devices under it exposed some interesting problems that apply to other devices besides dock stations. Right now we require the use of the pci=assign-buses parameter, mainly because the BIOS may not reserve enough bus numbers for us to insert a new dock bridge and other buses behind it. I found a couple problems with how bus numbers are assigned during my work which required patches to the PCI core. In many ways the problems that are faced with implementing hot dock are directly applicable to hotplugging on servers. Therefore, it is valuable work to continue, even if only 3 people in the world still use a docking station. We do believe that docking station usage will rise as system vendors create more compelling uses for them.

Dock station hardware implementations can really vary. It's very common to have a P2P bridge located on the dock station, with a tree of devices underneath it, however, it isn't the only implementation. Because of this, it's important to handle docking separately from any hotplug activity, so that all the intelligence for hotplug can be handled by the individual subsystems or drivers rather than in one gigantic dock driver. I have only implemented changes to allow one driver to hotplug after a dock, but more drivers or subsystems may be modified in the future.

I have very limited testing done at this point, and every time a new person tries the dock patches, the design must be modified to handle yet another hardware implementation. As usage increases, I expect that the implementation described in this paper will evolve to something which hopefully allows more and more laptop docking stations to "just work" with Linux.

References

[1] *Advanced Configuration and Power*

Interface specification. www.acpi.info, 3.0a edition.

[2] *PCI Hot Plug specification*. www.pcisig.com, 1.1 edition.

[3] *PCI Local Bus specification*. www.pcisig.com, 3.0 edition.

[4] *PCI-to-PCI Bridge specification*. www.pcisig.com, 1.2 edition.

[5] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers*. O'Reilly, http://lwn.net/Kernel/LDD3/.

Open Source Graphic Drivers—They Don't Kill Kittens

David M. Airlie
Open Source Contractor
airlied@linux.ie

Abstract

This paper is a light-hearted look at the state of current support for Linux / X.org graphics drivers and explains why closed source drivers are in fact responsible for the death of a lot of small cute animals.

The paper discusses how the current trend of using closed-source graphics drivers is affecting the Open Source community, e.g. users are claiming that they are running an open source operating system which then contains a 1 MB binary kernel module and 10MB user space module...

The paper finally look at the current state of open source graphics drivers and vendors and how they are interacting with the kernel and X.org communities at this time (this changes a lot). It discusses methods for producing open source graphics drivers such as the r300 project and the recently started NVIDIA reverse engineering project.

1 Current Official Status

This section examines the current status (as of March 2006) of the support from various manufactures for Linux, X.org[4], and DRI[1] projects. The three primary manufacturers, Intel, ATI, NVIDIA, are looked at in depth along with a brief overview of other drivers.

1.1 Intel

Currently Intel contract Tungsten Graphics to implement drivers for their integrated graphics chipsets. TG directly contribute code to the Linux kernel, X.org, and Mesa projects to support Intel cards from the i810 to the i945G. Intel have previously released complete register and programmer's reference guides for the i810 chipset; however, from the i830 chipset onwards, no information was given to external entities with the exception of Tungsten.

As of March 2006, all known Intel chipsets are supported by the i810 X.org driver.

1.1.1 2D

The Intel integrated chipsets vary in the number and type of connectable devices. The desktop chipsets commonly only have native support for CRTs, and external devices are required to drive DVI or tv-out displays. These external devices are connected to the chipset using either the DVO (digital video output) on i8xx, or sDVO (serial digital video output) on i9xx. These external devices are controlled over an i2c bus. The mobile chipsets allow for an inbuilt LVDS controller and sometimes in-built tv-out controller.

Due to the number of external devices available from a number of manufacturers (e.g. Silicon Image, Chrontel), writing a driver is a lot of hard work without a lot of manufacturer datasheets and hardware. The Intel BIOS supports a number of these devices.

For this reason the driver uses the Video BIOS (VBE) to do mode setting on these chipsets. This means that unless the BIOS has a mode pre-configured in its tables, or a mode is hacked in (using tools like i915resolution), the driver cannot set it. This stops the driver being used properly on systems where the BIOS isn't configured properly (widescreen laptops) or the BIOS doesn't like the mode from the monitor (Dell 2005FPW via DVI).

1.1.2 3D

The Intel chipsets have varying hardware support for accelerating 3D operations. Intel "distinguish" themselves for other manufacturers by not putting support for TNL in hardware, preferring to have optimized drivers do that stuff in software. The i9xx added support for HW vertex shaders; however, fragment shading is still software-based. The 3D driver for the Intel chipsets supports all the features of the chipset with the exception of *Zone Rendering*, a tile-based rendering approach. Implementing zone rendering is a difficult task which changes a lot of how Mesa works, and the returns are not considered great enough yet.

However, in terms of open source support for 3D graphics, Intel provide by far the best support via Tungsten Graphics.

1.2 ATI

ATI, once a fine upstanding citizen of then open source world (well they got paid for it), no longer have any interest in our little adventures and have joined the kitten killers. The ATI cards can be broken up into 3 broad categories, pre-r300, r3xx-r4xx, and r5xx.

1.2.1 pre-r300

Thanks to the Weather Channel wanting open-source drivers for the Radeon R100 and R200 family of cards, and paying the money, ATI made available cut-down register specifications for these chipsets to the open-source developer community via their Developer Relations website. These specifications allowed Tungsten Graphics to implement basically complete 3D drivers for the r100 and r200 series of cards. However, ATI didn't provide any information on programming any company-proprietary features such as Hyper-Z or TruForm. ATI's engineering also provided some support over a number of years for 2D on these cards, such as initial render acceleration, and errata for many chips.

1.2.2 r300–r4xx

The R300 marked the first chipset that ATI weren't willing to provide any 3D support for their cards. A 2D register spec and development kit was provided to a number of developers, and errata support was provided by ATI engineering. However, no information on the 3D sections of this chipset were ever revealed to the open source community. A number of OEMs have been provided information on these cards, but no rights to use it for open-source work.

ATI's fglrx closed-source driver appeared with support for many of the 3D features on the cards; it, however, has had certain stability problems and later versions do not always run on older cards.

This range of cards also saw the introduction of PCI Express cards. Support for these cards came quite late, and a number of buggy fglrx releases were required before it was stabilised.

1.2.3 r5xx

The R5xx is the latest ATI chipset. This chipset has a completely redesigned mode setting, and memory controller compared to the r4xx, the 3D engine is mostly similiar. Again no information has been provided to the open-source community. As of this writing no support beyond vesa is available for these chipsets. ATI have not released an open-source 2D driver or a version of fglrx that supports these chips, making them totally useless for any Linux users.

1.2.4 fglrx

The ATI fglrx driver supports r200, r300, and r400 cards, and is built using the DRI framework. It installs its own libGL (the DRI one used to be insufficent for their needs) and a quite large kernel module. FGLRX OpenGL support can sometimes be a bit useless, Doom3 for example crashes horribly on fglrx when it came out first.

1.3 NVIDIA

Most people have thought that NVIDIA were always evil and never provided any specs, which isn't true. Back in the days of the riva chipsets, the Utah-GLX project implemented 3D support for the NVIDIA chipsets using NVIDIA-provided documentation.

1.3.1 2D

NVIDIA have some belief in having a driver for their chipsets shipped with X.org even if it only supports basic 2D acceleration. This at least allows users to get X up and running so they can download the latest binary driver for their cards, or at least use X on PPC and other non-x86 architectures. The nv driver in X.org is supported by NVIDIA employees and despite it being written in obfuscated C}^Hhex code, the source is there to be tweaked. BeOS happens to have a better open source NVIDIA driver with dual-head support, which may be ported to X.org at some point.

1.3.2 3D

When it comes to 3D, the NVIDIA 3D drivers are considered the best "closed-source" drivers. From an engineering point of view, the drivers are well supported, and NVIDIA interact well with the X.org community when it comes to adding new features. The NVIDIA driver provides support for most modern NVIDIA cards; however, they recently dumped support for a lot of older cards into a legacy driver and are discontinuing support in the primary driver. NVIDIA drivers commonly support all features of OpenGL quite well.

1.4 Others

1.4.1 VIA and SiS

Other manufactures of note are Matrox, VIA, and SiS. VIA and SiS both suffer from a serious lack of interaction with the open-source community, most likely due to some cultural differences between Taiwanese manufacturers and open-source developers. Both companies occasionally code-drop drivers for hardware with bad license files, no response to feedback, but with a nice shiny press release or get in touch with open-source driver writers with promises of support and NDA'd documentation, but nothing ever comes of it. Neither company has a consistent approach to open source drivers. VIA chipsets have good support for features thanks to people taking their code drops and making proper X.org drivers from them (unichrome and openchrome projects), and SiS chipsets (via Thomas Winischofer) have probably by far the best 2D driver available in terms of features, but their 3D drivers are a bit hit-and-miss, and only certain chipsets are supported at all.

1.4.2 Matrox

Matrox provide open source drivers for their chipsets below G550; however, newer chipsets use a closed-source driver.

2 Closed Source Drivers—Reasons

So a question the author is asked a lot is why he believes closed source drivers are a bad thing. I don't consider them bad so much as pure evil, in the kitten-killing, seal-clubbing sense. Someone has to hold an extreme view on these things, and in the graphics driver case that is the author's position. This sections explores some of the reasons why open-source drivers for graphics card seems to be going the opposite direction to open-source drivers for every other type of hardware.

This is all the author's opinion and doesn't try to reflect truth in any way.

2.1 Reason—Microsoft

The conspiracy theorists among us (I'm not a huge fan), find a way to blame Microsoft for every problem in Linux. So to keep them happy, I've noticed two things.

- Microsoft decided to use a vendor's chip in the XBOX series → no specs anymore.
- Chipset vendors puts DirectX 8.0 support into a chip → no specs anymore.

Hope this keeps that section happy.

2.2 Reason—???

Patents and fear of competitors or patent scumsucking companies bringing infringement against the latest chipset release and delaying it, is probably a valid fear amongst chip manufacturers. They claim releasing chipset docs to the public may make it easier for these things to be found; however, most X.org developers have no problem signing suitable NDAs with manufacturers to access specs. Open source drivers may show a company's hand to a greater degree. This is probably the most valid fear and it is getting more valid due to the great U.S. patent system.

2.3 Reason—Profit

Graphics card manufacturing is a very competitive industry, especially in the high-end gaming, 3–6 month development cycle, grind-out-as-many-different-cards-as-you-can world that ATI and NVIDIA inhabit. I can't see how open sourcing drivers would slow down these cycles or get in the way—apart from the fact that the dirty tricks to detect and speed up quake 3 might be spotted easier (everyone spots them in the closed source drivers anyways). It doesn't quite explain Matrox and those guys who don't really engage in the gamer market to any great degree. It also doesn't really explain fglrx which are some of the most unsuitable drivers for gaming on Linux.

Also things like SLI and Crossfire bring to question some of the profit motivation; the number of SLI and Crossfire users are certainly less than the number of Linux users.

3 Closed Source Drivers—Killing Kittens

3.1 Fluffy—Open Source OS

Linux is an open source OS. Linux has become a highly stable OS due to its open source nature. The ability for anyone to be able to fix a bug in any place in the OS, within reason, is very useful for implementing Linux in a lot of server and embedded environments. Things like Windows CE work for embedded systems as long as you do what MS wanted you to do; Linux works in these systems because you don't have to follow the plan of another company: you are free to do your own things. Closed source drivers take this freedom away.

If you load a 1MB binary into your Linux kernel or X.org, you are **NO LONGER RUNNING AN OPEN SOURCE OS**. Lots of users don't realise this, they tell their friends all about open source, but use NVIDIA drivers.

3.2 Mopsy—Leeching

So on to why the drivers are a bad thing. Linux developers have developed a highly stable OS and provide the source to it, X.org is finally getting together a window system with some modern features in it and are providing the source to it. These developers are also providing the ideas and infrastructure for these things openly. Closed source vendors are just not contributing to the pool of knowledge and features in any decent ways. Open source developers are currently implementing acceleration architectures and memory management systems that the closed source drivers have had for a few years. These areas aren't exactly the family jewels, surely some code might have been contributed or some ideas on how things might be done.

3.3 Kitty—niche systems

There are a lot of niche systems out there, installations in the thousands that normally don't interest the likes of NVIDIA or ATI. The author implements embedded graphics systems, and originally use ATI M7s but now uses Intel chipsets where possible. These sales, while not significant to ATI or NVIDIA on an individual basis, add up to a lot more than the SLI or CrossFire sales ever will. However these niche systems usually require open source drivers in order to do something different. For example, the author's systems require a single 3D application but not an X server. Implementing this is possible using open source drivers; however, doing so with closed source driver is not possible. Also, for non-x86 systems such as PPC or Sparc, where these chips are also used, getting a functional driver under Linux just isn't possible.

3.4 Spot—out-dated systems

Once a closed vendor has sold enough of a card, it's time to move on and force people somehow to buy later cards. Supporting older cards is no longer a priority or profitable. This allows them to stop support for these cards at a certain level and not provide any new features on those cards even if it possible. Looking at the features added to the open-source radeon driver since its inception shows that continuing development on these cards is possible in the open source community. NVIDIA recently relegated all cards before a certain date to their legacy drivers. Eventually these drivers will probably stop being updated, meaning running a newer version of Linux on those systems will become impossible.

4 Open Source Drivers—Future

This section discusses the future plans of open source graphic driver development. This paper was written in March 2006, and a lot may have happened between now and the publishing date in July.[1] The presentation at the conference will hopefully have all-new information.

4.1 Intel

Going forwards, Intel appear to be in the best positions. Recent hirings show their support for open source graphics, and Tungsten Graphics have added a number of features to their drivers and are currently implementing an open source video memory manager initially on the Intel chipsets. Once the mode-setting issues are cleared up, the drivers will be the best example out there.

The author has done some work to implement BIOS-less mode setting on these cards for certain embedded systems, and hopes that work can be taken forward to cover all cards and integrated into the open source X.org driver and become supported by Intel/TG.

[1][Well, *besides* an awful lot of formatting, copyediting, and such—*OLS Formatting Team*.]

4.2 ATI

4.2.1 R3xx + R4xx 3D support

The R300 project is an effort to provide an open-source 3D driver for the r300 and greater by reverse engineering methods. The project has used the fglrx and Windows drivers to reverse engineer the register writes used by the r3xx cards. The method used involved running a simple OpenGL application and changing one thing at a time to see what registers were written by the driver. There are still a few problems with this approach in terms of stability, as certain card setup sequences for certain cards are not yet known (radeon 9800s fall over a lot). These sequences are not that easy to discover; however, tracing the fglrx startup using valgrind might help a lot.

While this project has been highly successful in terms of implementing the feature set of the cards, the lack of documentation and/or engineering support hamper any attempts to make this a completely stable system.

4.2.2 R5xx 2D support

A 2D driver for the R5xx series of cards from the author may appear; however, the author would like to engage ATI so as to avoid getting sued into the ground, due to a lot of information being available under NDA via an OEM. Most of the driver has, however, been reverse engineered by tracing the outputs from the video bios when asked to set a mode, using a modified x86 emulator.

4.3 NVIDIA

Recently an X.org DRI developer (Stephane Marcheu) announced the *renoveau project* [3], an attempt to build open-source 3D drivers for NVIDIA cards. This project will use the same methods as the r300 project to attempt to get at first a basic 3D driver for the NVIDIA cards.

5 Reverse Engineering Methodologies

This section just looks at some of the commonly used reverse engineering methodologies in developer graphics drivers.

5.1 2D Modesetting

Most cards come with a video BIOS that can set modes. Using a tool like LRMI[2], the Linux Real Mode Interface, the BIOS can be run inside an emulator. When the BIOS uses an `inl` or `outl` instruction to write to the card, LRMI must actually do this for it, so these calls can be trapped and used to figure out the sequence of register writes necessary to set a particular mode on a particular chipset. Multiple runs can be used to track exactly where the mode information is emitted.

This method has been used by the author in writing mode-setting code for Intel i915 chipsets, for intelfb and X.org, and also for looking at the R520 mode setting.

Another method, if a driver exists for a card under Linux already: a number of developers have discussed using an mmap trick, whereby a framework is built which loads the driver, and fakes the mmap for the card registers. The framework then catches all the segmentation faults and logs them while passing them through. This has been used by Ben Herrenschmidt for radeonfb suspend/resume support on Apple drivers. An enhancement to this by the author (who was too lazy to write an mmap framework for x86) uses a valgrind plugin to track the mmap and read/writes to an mmaped area. This solution isn't perfect (it only allows reading back writes after they happen), but it has been sufficent for work on the i9xx reverse engineering.

5.2 3D

Most 3D cards have some form of two-section drivers, a kernel-space manager and a userspace 3D driver linked into the application via libGL. The kernel-space code normally queues up command buffers from the userspace driver. The userspace driver normally mmaps the command queues into its address space. An application linked with libGL can do some simple 3D operations and then watch the command buffers as the app fills them. Tweaking what the highlevel application does allows different command buffers to be compared and a map of card registers vs. features can be built up. The r300 project has been very successful with this approach.

The r300 project also have a tool that runs under Windows, that constantly scans the shared buffers used by the windows drivers, and dumps them whenever they change in order to do similiar work.

6 Conclusion

This paper has looked at the current situation with graphics driver support for Linux and

X.org from card manufacturers. It looks at why closed source drivers are considered evil and looks at what the open source community is doing to try and provide drivers for open source cards. Just remember, save those kittens.

References

[1] DRI Project.
`http://dri.freedesktop.org`.

[2] Linux Real Mode Interface.
`http://lrmi.sf.net`.

[3] The Renoveau Project.
`http://renoveau.sf.net/`.

[4] The X.org Project.
`http://www.x.org/`.

kboot—A Boot Loader Based on Kexec

Werner Almesberger
werner@almesberger.net

Abstract

Compared to the "consoles" found on traditional Unix workstations and mini-computers, the Linux boot process is feature-poor, and the addition of new functionality to boot loaders often results in massive code duplication. With the availability of kexec, this situation can be improved.

kboot is a proof-of-concept implementation of a Linux boot loader based on kexec. kboot uses a boot loader like LILO or GRUB to load a regular Linux kernel as its first stage. Then, the full capabilities of the kernel can be used to locate and to access the kernel to be booted, perform limited diagnostics and repair, etc.

1 Oh no, not another boot loader !

There is already no shortage of boot loaders for Linux, so why have another one ? The motivation for making kboot is simply that the boot process of Linux is still not as good as it could be, and that recent technological advances have made it comparably easy to do better.

Looking at traditional Unix servers and workstations, one often finds very powerful boot environments, offering a broad choice of possible sources for the kernel and other system files to load. It is also quite common to find various tools for hardware diagnosis and system software repair. On Linux, many boot loaders are much more limited than this.

Even boot loaders that provide several of these advanced features, like GRUB, suffer from the problem that they need to replicate functionality or at least include code found elsewhere, which creates an ever increasing maintenance burden. Similarly, any drivers or protocols the boot loader incorporates, will have to be maintained in the context of that boot loader, in parallel with the original source.

New boot loader functionality is not only required because administrators demand more powerful tools, but also because technological progress leads to more and more complex mechanisms for accessing storage and other devices, which a boot loader eventually should be able to support.

It is easy to see that a regular Linux system happens to support a superset of all the functionality described above.

With the addition of the kexec system call to the 2.6.13 mainline Linux kernel, we now have an instrument that allows us to build boot loaders with a fully featured Linux system, tailored according to the needs of the boot process and the resources available for it.

Kboot is a proof-of-concept implementation of such a boot loader. It demonstrates that new functionality can be merged from the vast code base available for Linux with great ease, and without incurring any significant maintenance overhead. This way, it can also serve as a platform for the development of new boot concepts.

The project's home page is at `http://kboot.sourceforge.net/`

The remainder of this section gives a high-level view of the role of a boot loader in general, and what kboot aims to accomplish. Additional technical details about the boot process, including tasks performed by the Linux kernel when bringing up user space, can be found in [1].

Section 2 briefly describes Eric Biederman's kexec [2], which plays a key role in the operation of kboot. Section 3 introduces kboot proper, explains its structure, and discusses its application. Section 4 gives an outlook on future work, and we conclude with section 5.

1.1 What a boot loader does

After being loaded by the system's firmware, a boot loader spends a few moments making itself comfortable on the system. This includes loading additional parts, moving itself to other memory regions, and establishing access to devices.

After that, it typically tries to interact with the user. This interaction can range from checking whether the user is trying to get the boot loader's attention by pressing some key, through a command line or a simple full-screen menu, to a lavish graphical user interface.

Whatever the interface may be, in the end its main purpose is to allow the user to select, perhaps along with some other options, which operating system or kernel will be booted. Once this choice is made, the boot loader proceeds to load the corresponding data into memory, does some additional setup, e.g., to pass parameters to the operating system it is booting, and transfers control to the entry point of the code it has loaded.

In the case of Linux, two items deserve special mention: the boot parameter line and the initial RAM disk.

The boot parameter line was at its inception intended primarily as a means for passing a "boot into single user mode" flag to the kernel, but this got a little out of hand, and it is nowadays often used to pass dozens if not hundreds of bytes of essential configuration data to the kernel, such as the location of the root file system, instructions for how certain drivers should initialize themselves (e.g., whether it is safe for the IDE driver to try to use DMA or not), and the selection and tuning of items included in a generic kernel (e.g., disabling ACPI support).

Since a kernel would often not even boot without the correct set of boot parameters, a boot loader must store them in its configuration, and pass them to the kernel without requiring user action. At the same time, users should of course be able to manually set and override such parameters.

The initial RAM disk (initrd), which at the time of writing is gradually being replaced by the initial RAM file system (initramfs), provides an early user space, which is put into memory by the boot loader, and is thus available even before the kernel is fully capable to interact with its surroundings. This early user space is used for extended setup operations, such as the loading of driver modules.

Given that the use of initrd is an integral part of many Linux distributions, any general-purpose Linux boot loader must support this functionality.

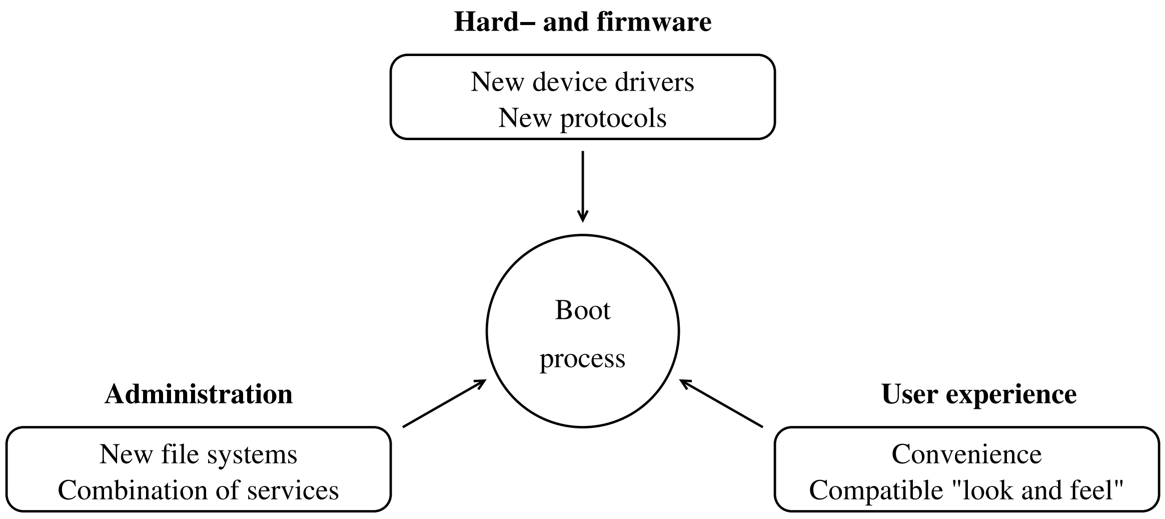


Figure 1: The boot process exists in a world full of changes and faces requirements from many directions. All this leads to the need to continuously grow in functionality.

1.2 What a boot loader should be like

A boot loader has much in common with the operating system it is loading: it shares the same hardware, exists in the same administrative context, and is seen by the same users. From all these directions originate requirements on the boot process, as illustrated in figure 1.

The boot loader has to be able to access at least the hardware that leads to the locations from which data has to be loaded. This does not only include physical resources, but also any protocols that are used to communicate with devices. Firmware sometimes provides a set of functions to perform such accesses, but new hardware or protocol extensions often require support that goes beyond this. For example, although many PCs have a Firewire port, BIOS support for booting from storage attached via Firewire is not common.

Above basic access mechanisms lies the domain of services the administrator can combine more or less freely. This begins with file system formats, and gets particularly interesting when using networks. For example, there is nothing inherently wrong in wanting to boot kernels that happen to be stored in RPM files on an NFS server, which is reached through an IPsec link.

The hardware and protocol environment of the boot process extends beyond storage. For example, keyboard or display devices for users with disabilities may require special drivers. With kboot, such devices can also be used to interact with the boot loader.

Last but not least, whenever users have to perform non-trivial tasks with the boot loader, they will prefer a context similar to what they are used to from normal interaction with the system. For instance, path names starting at the root of a file system hierarchy tend to be easier to remember than device-local names prefixed with a disk and partition number.

In addition to all this, it is often desirable if small repair work on an unbootable system can be done from the boot loader, without having to find or prepare a system recovery medium, or similar.

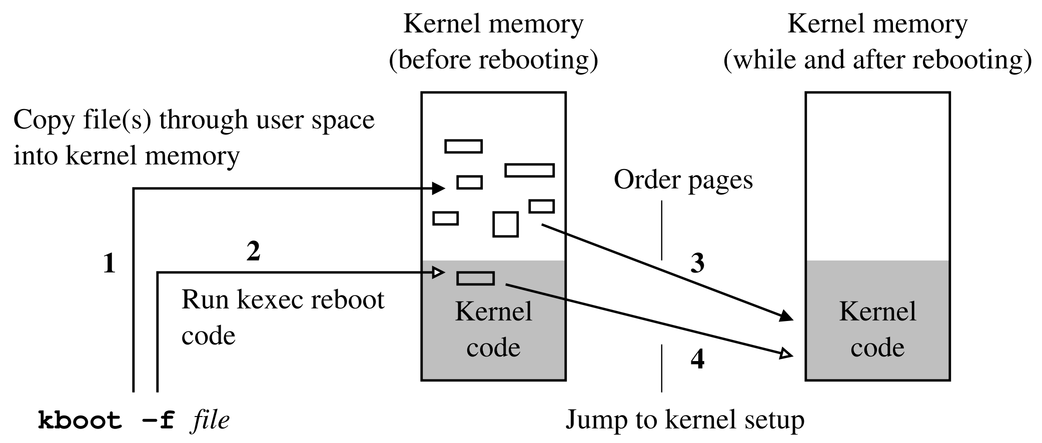


Figure 2: Simplified boot sequence of kexec.

The bottom line is that a general-purpose boot loader will always grow in functionality along the lines of what the full operating system can support.

1.3 The story so far

The two principal boot loaders for Linux on the i386 platform, LILO and GRUB, illustrate this trend nicely.

LILO was designed with the goal in mind of being able to load kernels from any file system the kernel may support. Other functionality has been added over time, but growth has been limited by the author's choice of implementing the entire boot loader in assembler. [1]

GRUB appeared several years later and was written in C from the beginning, which helped it to absorb additional functionality more quickly. For instance, GRUB can directly read a large number of different file system formats, without having to rely on external help, such as the map file used by LILO. GRUB also offers limited networking support.

Unfortunately, GRUB still requires that any new functionality, be it drivers, file systems, file formats, network protocols, or anything else, is integrated into GRUB's own environment. This somewhat slows initial incorporation of new features, and, worse yet, leads to an increasing amount of code that has to be maintained in parallel with its counterpart in regular Linux.

In an ideal boot loader, the difference between the environment found on a regular Linux system and that in the boot loader would be reduced to a point where integration of new features, and their subsequent maintenance, is trivial. Furthermore, reducing the barrier for working on the boot loader should also encourage customization for specific environments, and more experimental uses.

The author has proposed the use of the Linux kernel as the main element of a boot loader in [1]. Since then, several years have passed, some of the technology has first changed, then

[1]LILO was written in 1992. At that time, 32-bit real mode of the i386 processor was not generally known, and the author therefore had to choose between programming in the 16-bit mode in which the i386 starts, or implementing a fully-featured 32-bit protected mode environment, complete with real-mode callbacks to invoke BIOS functions. After choosing the less intrusive of the two approaches, there was the problem that no suitable and reasonably widely deployed free C compiler was available. Hence the decision to write LILO in assembler.

matured, and with the integration of the key element required for all this into the mainstream kernel, work on this new kind of boot loader could start in earnest.

2 Booting kernels with kexec

One prediction in [1] came true almost immediately, namely that major changes to the bootimg mechanism described there were quite probable: when Eric Biederman released kexec, it swiftly replaced bootimg, being technologically superior and also better maintained.

Unfortunately, adoption of kexec into the mainstream kernel took much longer than anyone expected, in part also because it underwent design changes to better support the very elegant kdump crash dump mechanism [3], and it was only with the 2.6.13 kernel that it was finally accepted.

2.1 Operation

This is a brief overview of the fundamental aspects of how kexec operates. More details can be found in [4], [5], and also [3].

As shown in figure 2, the user space tool `kexec` first loads the code of the new kernel plus any additional data, such as an initial RAM disk, into user space memory, and then invokes the `kexec_load` system call to copy it into kernel memory (1). During the loading, the user space tool can also add or omit data (e.g., setup code), and perform format conversions (e.g., when reading from an ELF file).

After that, a `reboot` system call is made to boot the new kernel (2). The reboot code tries to shut down all devices, such that they are in a defined and inactive state, from which they can be instantly reactivated after the reboot.

Since data pages containing the new kernel have been loaded to arbitrary physical locations and could not occupy the same space as the code of the old kernel before the reboot anyway, they have to be moved to their final destination (3).

Finally, the reboot code jumps to the entry point of the setup code of the new kernel. That kernel then goes through its initialization, brings up drivers, etc.

2.2 Debugging

The weak spot of kexec are the drivers: some drivers may simply ignore the request to shut down, others may be overzealous, and deactivate the device in question completely, and some may leave the device in a state from which it cannot be brought back to life, be this either because the state itself is incorrect or irrecoverable, or because the driver simply does not know how to resume from this specific state.

Failure may also be only partial, e.g., VGA often ends up in a state where the text does not scroll properly until the card is reset by loading a font.

Many of these problems have not become visible yet, because those drivers have not been subjected to this specific shutdown and reboot sequence so far.

The developers of kexec and kdump have made a great effort to make kexec work with a large set of hardware, but given the sheer number of drivers in the kernel and also in parallel trees, there are doubtlessly many more problems still awaiting discovery.

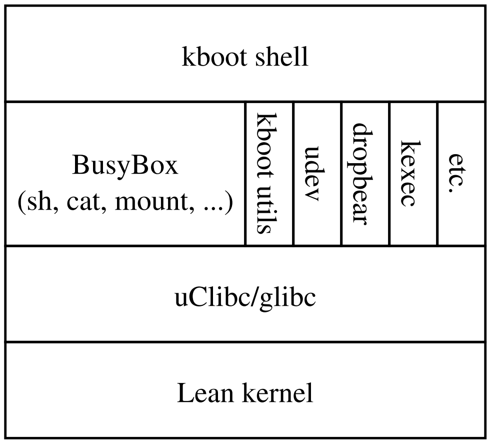


Figure 3: The software stack of the kboot environment.

Since kboot is the first application of kexec that should attract interest from more than a relatively small group of developers, many of the expected driver conflicts will surface in the form of boot failures occurring under kboot, after which they can be corrected.

3 Putting it all together

Kboot bundles the components needed for a boot loader, and provides the "glue" to hold them together. For this, it needs very little code: as of version 10, only roughly 3'500 lines, about half of this shell scripts. Already LILO exceeds this by one order of magnitude, and GRUB further doubles LILO's figure.[2]

Of course, during its build process, kboot pulls in various large packages, among them the entire GCC tool chain, a C library, BusyBox, assorted other utilities, and the Linux kernel itself. In this regard, kboot resembles more a distribution like Gentoo, OpenEmbedded, or Rock Linux, which consist mainly of meta-information about packages maintained by other parties.

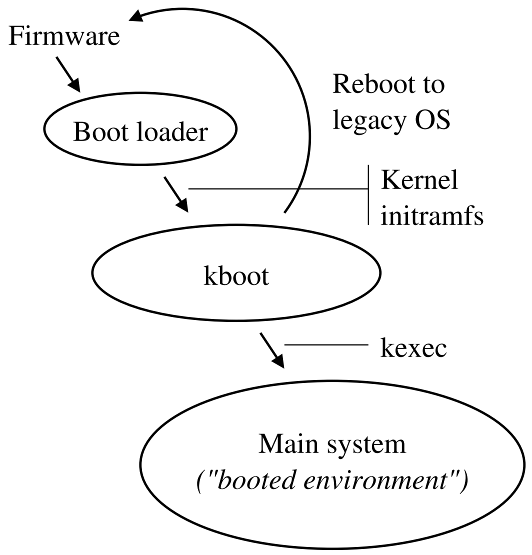


Figure 4: The boot sequence when using kboot.

3.1 The boot environment

Figure 3 shows the software packages that constitute the kboot environment. Its basis is a Linux kernel. This kernel only needs to support the devices, file systems, and protocols that will be used by kboot, and can therefore, if space is an issue, be made considerably smaller than a fully-featured production kernel for the same machine.

In order to save space, kboot can use uClibc [6] instead of the much larger glibc. Unfortunately, properly supporting a library different from the one on the host system requires building a dedicated version of GCC. Since uClibc is sensitive to the compiler version, kboot also builds a local copy of GCC for the host. To be on the safe side, it also builds binutils.

After this tour de force, kboot builds the applications for its user space, which include BusyBox [7], udev [8], the kexec tools [2], and

[2]These numbers were obtained by quite unscientifically running `wc-l` on a somewhat arbitrary set of the files in the respective source trees.

dropbear [9]. BusyBox provides a great many common programs, ranging from a Bourne shell, through system tools like "mount," to a complete set of networking utilities, including "wget" and a DHCP client. Udev is responsible for the creation of device files in `/dev`. It is a user space replacement for the kernel-based devfs. The kexec tools provide the user space interface to kexec.

Last but not least, dropbear, an SSH server and client package, is included to demonstrate the flexibility afforded by this design. This also offers a simple remote access to the boot prompt, without the need to set up a serial console for just this purpose.

3.2 The boot sequence

The boot sequence, shown in figure 4, is as follows: first, the firmware loads and starts the first-stage boot loader. This would typically be a program like GRUB or LILO, but it could also be something more specialized, e.g., a loader for on-board Flash memory. This boot loader then immediately proceeds to load kboot's Linux kernel and kboot's initramfs.

The kernel goes through the usual initialization and then starts the kboot shell, which updates its configuration files (see section 3.5), may bring up networking, and then interacts with the user.

If the user chooses, either actively or through a timeout, to start a Linux system, kboot then uses kexec to load the kernel and maybe also an initial RAM disk.

Although not yet implemented at the time of writing, kboot will also be able to boot legacy operating systems. The plan is to initially avoid the quagmire of restoring the firmware environment to the point that the system can be booted from it, but to hand the boot request back to the first stage boot loader (e.g., with `lilo-R` or `grub-set-default`), and to reboot through the firmware.

3.3 The boot shell

At the time of writing, the boot shell is fairly simple. After initializing the boot environment, it offers a command line with editing, command and file name completion, and a history function for the current session.

The following types of items can be entered:

- Names of variables containing a command. These variables are usually defined in the kboot configuration file, but can also be set during a kboot session.[3] The variable is expanded, and the shell then processes the command. This is a slight generalization of the `label` in LILO, or the `title` in GRUB.
- The path to a file containing a bootable kernel. Path names are generalized in kboot, and also allow direct access to devices and some network resources. They are described in more detail in the next section. When such a path name is entered, kboot tries to boot the file through kexec.
- The name of a block device containing the boot sector of a legacy operating system, or the path to the corresponding device file.
- An internal command of the kboot shell. It currently supports `cd` and `pwd`, with the usual semantics.

[3] In the latter case, they are lost when the session ends.

Syntax	**Example**	**Description**
variable	`my_kernel`	Command stored in a variable
/*path*	`/boot/bzImage-2.6.13.2`	Absolute path in booted environment
//*path*	`cat //etc/fstab`	Absolute path in kboot environment
path	`cd linux-2.6.14`	Relative path in current environment
device	`hda7`	Device containing a boot sector
/dev/*device*	`/dev/hda7`	Device file of device with boot sector
device:/*path*	`hda1:/bzImage`	File or directory on a device
device:*path*	`hda1:bzImage`	(implicit /dev/)
/dev/*device*:/*path*	`/dev/sda6:/foo/bar`	File or directory on a device
/dev/*device*:*path*	`/dev/sda6:foo/bar`	(explicit /dev/)
host:/*path*	`server:/home/k/bzImage-a`	File or directory on an NFS server
http://*host*/*path*	`http://server/foo`	File on an HTTP server
ftp://*host*/*path*	`ftp://server/foo/bar`	File on an FTP server

Table 1: Types of path names recognized by kboot.

- A shell command. The kboot shell performs path name substitution, and then runs the command. If the command uses an executable from the booted environment, it is run with chroot, since the shared libraries available in the kboot environment may be incompatible with the expectations of the executable.

With the exception of a few helper programs, like the command line editor, the kboot shell is currently implemented as a Bourne shell script.

3.4 Generalized path names

Kboot automatically mounts file systems of the booted environment, on explicitly specified block devices, and—if networking is enabled—also from NFS servers. Furthermore, it can copy and then boot files from HTTP and FTP servers.

For all this, it uses a generalized path name syntax that reflects the most common forms of specifying the respective resources. E.g., for NFS, the *host*:*path* syntax is used, for HTTP, it is a URL, and paths on the booted environment look just like normal Unix path names. Table 1 shows the various forms of path names.

Absolute paths in the kboot environment are an exception: they begin with two slashes instead of one.

We currently assume that there is one principal booted system environment, which defines the "normal" file system hierarchy on the machine in question. Support for systems with multiple booted environments is planned for future versions of kboot.

3.5 Configuration files

When kboot starts, it only has access to the configuration files stored in its initramfs. These were gathered at build time, either from the user (who placed them in kboot's `config/` directory), or from the current configuration of the build host.

This set of files includes kboot's own configuration `/etc/kboot.conf`, `/etc/fstab`,

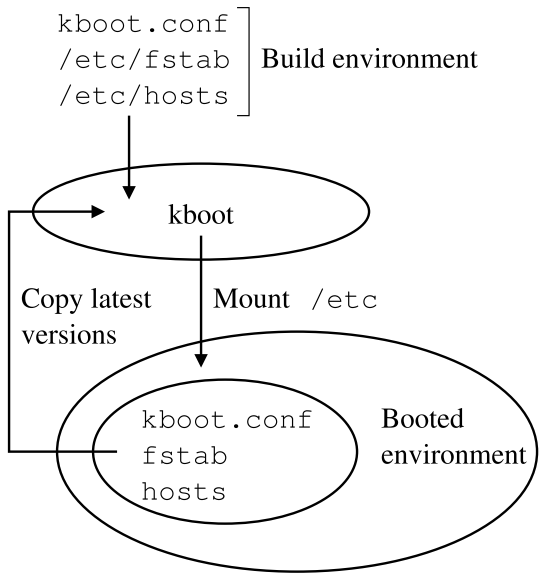


Figure 5: Some of the configuration files used by kboot.

and `/etc/hosts`. The kboot build process also adds a file `/etc/kboot-features` containing settings needed for the initialization of the kboot shell.

Kboot can now either use these files, or it can, at the user's discretion, try to mount the file system containing the `/etc` directory of the booted environment, and obtain more recent copies of them.

The decision of whether kboot will use its own copies, or attempt an update first, is made at build time. It can be superseded at boot time by passing the kernel parameter `kboot=local`.

3.6 When not to use kboot

While kboot it designed to be a flexible and extensible solution, there are areas where this type of boot loader architecture does not fit.

If only very little persistent storage is available, which is a common situation in small embedded systems, or if large enough storage devices would be available, but cannot be made an integral part of the boot process, e.g., removable or unreliable media, only a boot loader optimized for tiny size may be suitable.

Similarly, if boot time is critical, the time spent loading and initializing an extra kernel may be too much. The boot time of regular desktop or server type machines already greatly exceeds the minimum boot time of a kernel, which embedded system developers aim to bring well below one second [10], so loading another kernel does not add too much overhead, particularly if the streamlining proposed below is applied.

Finally, the large hidden code base of kboot is unsuitable if high demands on system reliability, at least until the point when the kernel is loaded, require that the number of software components be kept to a minimum.

3.7 Extending kboot

The most important aspect of kboot is not the set of features it already offers, but that it makes it easy to add new ones.

New device drivers, low-level protocols (e.g., USB), file systems, network protocols, etc., are usually directly supported by the kernel, and need no or only little additional support from user space. So kboot can be brought up to date with the state of the art by a simple kernel upgrade.

Most of the basic system software runs out of the box on virtually all platforms supported by Linux, and particularly distributions for embedded systems provide patches that help with the occasional compatibility glitches. They also maintain compact alternatives to packages where size may be an issue.

Similarly, given that kboot basically provides a regular Linux user space, the addition of new ornaments and improvements to the user interface, which is an area with a continuous demand for development, should be easy.

When porting kboot to a new platform, the foremost—and also technically most demanding—issue is getting kexec to run. Once this is accomplished, interaction with the boot loader has to be adapted, if such interaction is needed. Finally, any administrative tools that are specific to this platform need to be added to the kboot environment.

4 Future work

At the time of writing, kboot is still a very young program, and has only been tested by a small number of people. As more user feedback arrives, new lines of development will open. This section gives an overview of currently planned activities and improvements.

4.1 Reducing kernel delays

The Linux kernel spends a fair amount of time looking for devices. In particular, IDE or SCSI bus scans can try the patience of the user, also because they repeat similar scans already done by the firmware. The use of kboot now adds another round of the same.

A straightforward mechanism that should help to alleviate such delays would be to predict their outcome, and to stop the scan as soon as the list of discovered devices matches the prediction. Such a prediction could be made by kboot, based on information obtained from the kernel it is running under, and be passed as a boot parameter to be interpreted by the kernel being booted.

Once this is in place, one could also envision configuring such a prediction at the first stage boot loader, and passing it directly to the first kernel. This way, slow device scans that are known to always yield the same result could be completely avoided.

4.2 From shell to C

At the time of writing, the boot shell is a Bourne shell script. While this makes it easy to integrate other executables into the kboot shell, execution speed may become an issue, and also other language properties, such as the difficulty of separating name spaces, and how easily subtle quoting bugs may escape discovery, are turning into serious problems.

Rewriting the kboot shell in C should yield a program that is still compact, but easier to maintain.

4.3 Using a real distribution

The extensibility of kboot can be further increased by replacing its build process, which is very similar to that of buildroot [11], with the use of a modular distribution with a large set of maintained packages. In particular OpenEmbedded [12] and Rock Linux [13] look very promising.

The reasons for not reusing an existing build process already from the beginning were mainly that kboot needs tight control over the configuration process (to reuse kernel configuration, and to propagate information from there to other components) and package versions (in order to know what users will actually be building), the sometimes large set of prerequisites, and also problems encountered during trials.

4.4 Modular configuration

Adding new functionality to the kboot environment usually requires an extension of the build process and changes to the kboot shell. For common tasks, such as the addition of a new type of path names, it would be desirable to be able to just drop a small description file into the build system, which would then interface with the rest of kboot over a well-defined interface.

Regarding modules: at the time of writing, kboot does not support loadable kernel modules.

5 Conclusions

Kboot shows that a versatile boot loader can be built with relative little effort, if using a Linux kernel supporting kexec and a set of programs designed with the space constraints of embedded systems in mind.

By making it considerably easier to synchronize the boot process with regular Linux development, this kind of boot loader architecture should facilitate more timely support for new functionality, and encourage developers to explore new ideas whose implementation would have been considered too tedious or too arcane in the past.

References

[1] Almesberger, Werner. *Booting Linux: The History and the Future*, Proceedings of the Ottawa Linux Symposium 2000, July 2000. `http://www.almesberger.net/cv/papers/ols2k-9.ps`

[2] Biederman, Eric W. Kexec tools and patches. `http://www.xmission.com/~ebiederm/files/kexec/`

[3] Goyal, Vivek; Biederman, Eric W.; Nellitheertha, Hariprasad. *Kdump, A Kexec-based Kernel Crash Dumping Mechanism*, Proceedings of the Ottawa Linux Symposium 2005, vol. 1, pp. 169–180, July 2005. `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`

[4] Pfiffer, Andy. *Reducing System Reboot Time with kexec*, April 2003. `http://www.osdl.org/archive/andyp/kexec/whitepaper/kexec.pdf`

[5] Nellitheertha, Hariprasad. *Reboot Linux Faster using kexec*, May 2004. `http://www-128.ibm.com/developerworks/linux/library/l-kexec.html`

[6] Andersen, Erik. *uClibc*. `http://www.uclibc.org/`

[7] Andersen, Erik. *BUSYBOX*. `http://busybox.net/`

[8] Kroah-Hartman, Greg; *et al. udev*. `http://www.kernel.org/pub/linux/utils/kernel/hotplug/udev.html`

[9] Johnston, Matt. *Dropbear SSH server and client*. `http://matt.ucc.asn.au/dropbear/dropbear.html`

[10] CE Linux Forum. *BootupTimeResources*, CE Linux Public Wiki. `http://tree.celinuxforum.org/pubwiki/moin.cgi/BootupTimeResources`

[11] Andersen, Erik. *BUILDROOT*. `http://buildroot.uclibc.org/`

[12] *OpenEmbedded*. `http://oe.handhelds.org/`

[13] *Rock Linux*. `http://www.rocklinux.org/`

Ideas on improving Linux infrastructure for performance on multi-core platforms

Maxim Alt
Intel Corporation
maxim.alt@intel.com

Abstract

With maturing compiler technologies, compile-time analysis can be a very powerful tool for optimizing and monitoring code on any architecture. In combination with modern run-time analysis tools and existing program interfaces to monitor hardware counters, we will survey modern techniques for analyzing performance issues. We propose using performance counter data and sequences of performance events to trigger event handlers in either the application or the operating system. In this way, sequence of performance events can be your debugging breakpoint or a callback. This paper will try to bridge the capabilities of advanced performance monitoring with common software development infrastructure (debuggers, gcc, loader, process scheduler). One proposed approach is to extend the run-time environment with an interface layer that will filter performance profiles, capture sequences of performance hazards, and provide summary data to the OS, debuggers, or application.

With the introduction of hyper-threading technology several years ago, there were obvious challenges to look beyond a single running process to monitor and schedule compute intensive processes on multi-threaded cores. Multi-level memory hierarchy and scaling on SMP systems complicated the situation even further, causing essential changes in kernel scheduler and performance tools. In the era of parallel and platform computing, we rely less on single execution process performance—with each component optimized by the compiler—and it becomes important to evaluate performance of the platform as a whole. The new concept of performance adaptive schedulers is one example of intelligently maximizing the performance on platform level of CMP systems. Performance data at higher granularity and a concept of processor efficiency per functionality can be applied to making intelligent decisions on process scheduling in the operating system.

Towards the end, we will suggest particular improvements in performance and run-time tools as a reflection of proposed approaches and transition to platform-level optimization goals.

1 Introduction

This paper introduces a series of ideas for bringing together existing disparate technologies to improve tools for the detection and amelioration of performance hazards.

About 20 years ago, an exceptionally thin 16-bit real-time iRMX operating system had an extremely simple but important built-in feature: when debugging race conditions in shared memory at any point of run-time execution, a developer could bring the system to a halt, set an internal OS breakpoint at an address and the operating system would halt whenever a value was being written at the specified address. No high level debugger was needed to debug race conditions, nor were they capable.

With the introduction of hyper-threading technology, many software vendors started to experiment with running their multi-threaded software on hyper-threaded processors. With the increase the number of processors, those vendors expected to get good scaling after elimination of synchronization and scheduling issues. These issues are quite difficult to track, debug, or find with Gdb or VTune analyzer.

The problem of debugging synchronization issues in multi-threaded applications is growing more important and more complex with the advent of parallelizing compilers and language support for multithreading, such as OpenMP. The compiler has knowledge of program semantics, but does not generally have run-time information. The OS is in a position to make decisions based on run-time information, but it doesn't have the semantic information that was available to the compiler, since it sees only the binary. Further, any run-time analysis and decision-making affects application performance, either directly by using CPU time, or indirectly by effects such as cache pollution.

Spin locks are an example of the kind of construct for which higher-level information would be helpful. The compiler can easily detect spin lock constructs using simple pattern-recognition. From the run-time perspective a spin lock is a loop that repeatedly reads a shared memory address and compares it to a loop-invariant a value. A spin-lock is a useful synchronization mechanism if the stall is not long. For long stalls, it wastes a lot of processor time. Typically, after spinning for some number of cycles, the thread will yield to let other threads make progress. If the OS could identify which threads were waiting for a lock and which threads held the lock, it could adjust priorities to maximize throughput automatically.

Another relevant example of how a scheduler might use performance data is not based on debugging. Consider two processes running concurrently: two floating point intensive loops, where only one of which has long memory stalls. Should the scheduling of the processes alter? For example, two floating point intensive loops with unknown memory latency or two loops of unknown execution property or two blocks of code of unknown programming construct?

This question was raised by my colleagues in [1] about 3 years ago, where it was discussed how beneficial it would be if the OS scheduler had built-in micro-architectural intelligence.

Another issue with debugging the performance of an application using spin locks is that it typically doesn't provide much insight to know that the spin-lock library code is hot. The application programmer needs to know which lock is being held. That information can be gathered from the address of the lock, but often it is more useful to have a stack trace gathered at the time the lock is identified as hot. This requires sampling the return stack in the performance monitor handler, not just the current instruction pointer.

A process or a code block (hot block or block of interest) can be characterized by performance and code profiles, where the code profile is represented by a hierarchy of basic programming constructs, and the performance profile is represented by execution path along with registers image captured at any given point in time. In this paper we will describe a set of new profile-guided static and dynamic approaches for efficient run-time decisions: debugging, analyzing and scheduling.

Please refer to Appendix A for an overview of existing technologies, tools and utilities.

2 Bridging the Technologies

I would like to explore extending the scope of sampling-analysis hybrid tools for example by profiling with helper threads[10.5].

This section will provide a series of examples on how to combine building block components to get useful sample-analyze schemas, which could potentially turn into standalone tools:

- *A Pintool doing Pronto.* Given the non-existent ability of Pin to sample performance counters in optimize/analyze mode, a hybrid tool when Pronto is based on a Pintool would allow dynamic implementation of collecting performance data via pintool instrumentation. If sampling is needed then a sampling driver can be initiated and called from a pintool using PAPI. The PAPI interface allows you to start, stop, and read the performance counters (e.g. calls to PAPI_start and PAPI_stop using Pin's instrumentation interface). PAPI does require a kernel extension. This idea may also be implemented through a static HP's Caliper tool[1]

- *A Pintool to seek for hotspots.* Hotspot analysis can be done by defining "what it means to be a hotspot" by a pintool, or statically by parsing sampling data with scripts.

- *A Pintool which uses performance feedback data.* Theoretically, a Pintool could be built which uses performance data which has been collected (perhaps by some other tool) on previous runs. Intel has a file format for storing performance monitor unit data (".hpi" files) which are used by the compiler. Pintool reads events and performance counters dynamically as it executes a binary.

- *A Pintool to recognize event patterns.* Sequitur can be used for static or dynamic analysis (with a certain performance overhead) for complex grammars or in the context of this paper, event sequences.

2.1 Performance Overhead of *Sample-analyze* Workflow

Pintools instrumentation can be intrusive and the overhead is dependent upon the particular tool used. The generated code under pintool is the only code that executes, and the original code is kept for reference. Because instrumentation can perturb the performance behavior of the application, it is desirable to be able to collect performance data during an uninstrumented run, and use that data during a later instrumentation or execution run.

In addition, Sequitur performance for generic grammars containing many symbols may be extremely heavy[2].

2.2 Developers' Pain

As one could imagine, the multi-threaded application developers who are debugging and running on multi-core architectures need profiling tools (such as VTune analyzer or EMON) to be aware of the stack frame and the execution

[1]`http://h21007.www2.hp.com/dspp/tech/tech_TechSoftwareDetailPage_IDX/1,1703,1174,00.html` (currently available for Itanium microarchitecture only)

[2]Incorporation of the Sequitur algorithm into your instrumentation is an essential part of the techniques described in this paper. Due to the significant performance overhead, Sequitur is not used in generic form, and requires tailoring for particular usage cases with simplified grammar. It helps to find performance hazards and sequences of events of interest, or these, which characterizes the application process.

context. The tools also need to take into account procedure inlining. It would also be useful if the simple data derived out of these tools could be used by the run-time environment to adapt the environment for better throughput.

In this section we would consider code examples known to cause much pain when debugging or scheduling. Then, we will suggest ways to adjust the Linux infrastructure to leverage and integrate existing tools mentioned above to address these painful situations.

2.3 Profile-guided Debugging

A very common example is when you profile a multi-threaded application with frequent inter-process communications and interlocking. Many enterprise applications (web servers, data base servers, application servers, telecommunication applications, distributed content providers, etc.) suffer from complex synchronization issues when scaled. While optimizing such application with standard profiling tools, it is common to observe most of the cycles being spent in synchronization objects themselves. For example, wait for an object, idle loops, spin loops on shared memory.

Whether the implementation of synchronization objects is proprietary or via POSIX threads [15], a hot spot is noted as entering/leaving the critical section or locking/unlocking the shared object. Deeper analysis of such hotspots usually shows there is not much to optimize further on a micro-architectural level unless one is trying to optimize the performance of glibc. The real question is how to find out what objects actually originated a problematic idle or spin time in a millions-of-code-lines application with hundreds of locks? To track and instrument locks is not an easy task; it is similar to tracking memory allocations. Standard debugger techniques are not effective in identifying the underlying application issue.

```
Spinlock (mutex_t *m) {
Int I;
For (i=0; I < spin_count; i++)
  if (pthread_mutex_trylock(m) != EBUSY) return;
  pthread_mutex_lock(m); //or sometimes Sleep(M)
}
```

Figure 1: Spin lock

```
spin_start:
  pause
  Test  [mem], val  ; pre-read to save the
                    ; atomic op penalty
  J     Skip_xchg
  Lock  cmpxchg [mem], val ; shows bus serial-
                           ; ization stall

Skip_xchg:
  jnz   spin_start
```

Figure 2: Spin Lock Loop

A standard adaptive spin lock implementation looks similar to Figure 1 where inner spin lock loop translates to instructions shown in Figure 2.

Let's analyze what characterizes this code. One obvious implication of using atomic operations (for entering/leaving critical section it is atomic `add/dec`) is that such operations serialize the memory bus, which yields significant stalls due to pipeline flush and cache line invalidation for `[mem]`.

The code in Figure 2 would generate similar performance event patterns on most architectures. Following are the properties which characterize the code block profile:

- Very short loop (2–5 instructions)
- Very short loop containing `nop` or `rep nop` (`pause`)
- Contains instruction with Lock prefix yielding bus serialization
- Contains either `xchg` or `dec` or `add` instruction

The performance event profile for this block has the following properties:

- Likely branch misprediction at the 'loop' statement
- Very high CPI (cycles per instruction), as there is no parallelism possible
- Data bus utilization ratio (> 50%)
- Bus Not Ready ratio (> 0.01)
- Burst read contribution to data bus utilization (> 50%)
- Processor/Bus Writeback contribution to data bus utilization (> 50%)
- Parallelization ratio (< 1)
- Microops per instruction retired (> 2)
- Memory requests per instruction (> 0.1)
- Modified data sharing ratio (> 0.1)
- Context switches is high

We can define the grammar which consists of: loop length, loops with nops, locks, adds, dec, xchg; misprediction branches, high CPI, context switches, high data bus utilization.

It is necessary to quantify each of the performance counters' values (also called knobs) so we could establish a trigger for potential performance hazard. From the properties of spin lock block we can define a rule for a block to become the hot block or the block of interest. Then, similar to hot stream data prefetch example in Appendix A, we will use Sequitur to detect hot blocks containing event sequences within the defined grammar as you can see below in Figure 5

In order to simplify the problem for the spin lock detection, we may limit ourselves to the analysis of only code profile, as the entire performance profile is a direct result of having an instruction with a 'lock' prefix (e.g. a 'lock' prefix on any instruction results in data bus serialization, yielding known set of performance stalls). The 'spin lock' code properties can be dynamically obtained at run-time by instrumentation. We can use the Pin command line knob facility to define a block's heat. For example these knobs may be:

- Matched number of samples to trigger the hazard
- Number of consecutive samples
- Minimum and maximum length for hot block
- Minimum spins to consider it hot
- Maximum number of instructions to consider loop short

This technique[3] would allow us to insert a breakpoint on an event of performance hazard - the hot spin lock according to user's definition of a performance concern. The debugger would be able to stop and display stack frame of the context when the given sequence of events had occurred.[4]

The described dynamic mechanism (one of the suggested "sample-analyze" workflows) detects a block of interest and breaks the execution on a performance issue.

[3]For the spin lock profile, running the sampling along pintool instrumented executable is safe and correct, since main characteristics of spin lock could not be disrupted by instrumented code shown above

[4]Ideas for a standalone profiling tool - Assume we have an ability to improve an open source (PAPI-based) or proprietary (VTune analyzer) profiling tool. Instead of Int 3 insertion we could insert a macro operation to dump a stack frame and register contents by the sampling driver. The existing symbol table would allow tracking source-level performance hazards defined by the pintool's knobs.

```
// Run in optimize mode only - no need for sampling mode run

for (each basic block)
  for (each instruction in block) {
    if (Instruction is branch) {
      target = TargetAddress(ins);
      if (trace_start < target && target < address(ins) &&
         (target - address < short loop knob)) {
        Insert IfCall (trace_count--);
        Insert ThenCall (spin_count++);
        New grammar (knobs);
      }
    }

    if ((instruction in block has lock prefix) &&
       (instruction is either xchg, cmpxhg, add, dec)) {
      if (grammar->AddEvent(address(ins)) &&
         (block_heat++ > hot block knob)) {
        Insert Interrupt 3;        // for debugging the application
      }
    }
  }
```

Figure 3: Pintool's trace instrumentation pseudocode

For the static profiling schema we will modify this workflow as follows:

On run-time side, as follows:

- Pintool instrumentation inserting software-generated interrupts would stay the same as in dynamic case. Pintool would read the information about hot spin locks from static profiling results (PGO) or pronto repository

-Allow the debugger to read pronto repository directly. This data would contain pairs, such as (ip address, number of times the IP is reached - signifying the heat of the block)

On compile-time side:

- A newly developed pintool that would be similar in functionality to PGO and profrun utility without sampling. However this pintool would contain same detection algorithm by the Sequitur as described in Figure 3, which detects hot spin locks according to user defined knob values marking the heat. In this manner, pronto_tool is virtually replaced with the Sequitur. Upon detection of a hot block, the pintool spills the pair (ip, frequency) into the pronto repository.

- Due to unique code properties, the current implementation of PGO and profrun utility already contain the needed information about spin lock block's code profile. We still need to build a script which would replace analysis tool pronto_tool and is based on parsing profile data with the Sequitur. This mechanism would extract event patterns matching our definition of the spin lock block's heat. The detected pair (ip address, number of times this IP has been reached until block became hot) is inserted into profiling info, and subsequently passed to the debugger

This would summarize another suggestion on a new standalone run-time tool that reads in the profile data and use it to find hot locks, and then tells the debugger the IP of those blocks so it can stop there.

With our attempt to characterize code by its performance profile, one may ask how adequate the mapping between an actual code block and its performance profile is. Would a sequence of events spanned by performance properties symbolize a spin lock code, or in other words, how uniquely do code block properties define the code itself? For performance debugging or adaptation the functionality of a hot block itself is not important. Rather, what important is it's algorithm mapping on the micro-architecture and the stalls caused by this mapping. Therefore, it is sufficient to accurately describe a performance hazard and signal when its properties have occurred.

2.4 Performance Adaptive Run-time Scheduler

Consider running a high performance multi-threaded application. Many computation and memory intensive applications (rendering, encoding, signal processing, etc.) suffer from complex scheduling issues when scaling on modern multi-threaded and multi-processor architectures. Often, the developers optimize these applications by parallelizing single threaded computation to run multiple threads. In order to make the application run well in parallel, the developers perform functional decomposition.[5]

Then, the OS scheduler takes over the decision on how to schedule these functionally decomposed threads onto the available hardware. Since the OS scheduler is not aware of micro-architecture, functional decomposition, or OpenMP, parallelization often leads to performance degradation. In order to analyze this phenomenon there were many researches on informed multi-threaded schedulers [12], symbiotic job scheduling for SMP [13], [14], and MASA [1]. In this paper we will take an approach of bridging existing profiling tools and advanced compiler technologies to take a step further in solving this problem.

As an example, consider open source *LAME mp3 encoder*[6]. It is clear that the application is both computation and memory intensive, where computation is mostly floating point. Functional decomposition of hotspot function *lame_ecnode_mp3_frame()* is equivalent to a functional decomposition of *L3psycho_anal()* function. All decomposed threads at any point in time could unfold into a situation when running processes utilize similar resources on the same physical core (e.g. threads are: floating point intensive, floating point intensive, heavy integer computations, heavy integer computations, long memory latency operations, long memory latency operations).

As in the previous section, running a thread's profile consists of performance and code properties. Below, we will analyze such properties and the knobs defining the heat:

Following is the structure of properties for floating point operations intensive code block:

- Estimated functional imbalance originated by compiler's scheduler

- Estimated CPI by the compiler's scheduler

- Outer loop iteration count

[5]Functional decomposition is the analysis of the activity of a system as the product of a set of subordinate functions performed by independent subsystems, each with its own characteristic domain of application.

[6]`http://lame.sourceforge.net/download/download.html`

You can see similar characteristics in integer intensive and memory intensive code blocks. These code block properties ignore possible coding style inhibitors and are agnostic to some optimization techniques (such as code motion). Nested loops and non-inlined calls within a loop are being merged into single region at the run-time, since the block of interest in this case would be an outer block encapsulating multiple iterations to the same instruction pointer.

Event profile to determine performance properties for floating point intensive block:

- Balanced execution and parallelism – actual cycles per instruction ratio

- Microops per instruction retired for very long latency instructions (FP)

- FP assist and saturation event per retired FLOPs

- Retired FLOPs per relative number to instruction retired

- Conversion operations RTDC per relative number to instruction retired

- SSE instruction retired per instruction retired

Event profile for memory intensive block:

- Data bus utilization ratio (> 30%)

- Bus Not Ready ratio (> 0.001)

- Burst read contribution to data bus utilization (> 30%)

- Processor/Bus Writeback contribution to data bus utilization (> 30%)

- Microops per instruction retired (> 2) for repeat instructions

- Memory requests per instruction (> 0.1)

- Modified data sharing ratio (> 0.1)

In order to write a pintool for this topic, it is necessary to be able to deliver some compile-time derived data to the run-time. In particular, some code block characteristics can be easily determined by the compiler's scheduler: For example, CPI and other parallelism metrics, scheduled memory operations, scheduled floating point operations, etc. Compilers can output such information via object code annotations, optimizer reports, or post-compilation scripts that can strip required statistic on the generated assembly code. The recent changes in GCC's vectorizer and optimizer include Tree SSA[7]. It is possible to get the compiler scheduler's reports using `--ftree-vectorizer-verbose` compiler option. The code block properties derived from the compiler scheduler's data only needed on hot blocks. However, the compiler does not know which block is hot unless PGO or Pronto was used.

On the run-time side, Pin has a disassembly engine built-in. A pintool would be able to easily determine functional unit imbalance in a hot block if it isn't available from the compiler. Assuming that Pin has the ability to retrieve some compiler scheduler data, there are a few ways to create a pintool to determine whether running code has properties of floating point or memory intensive hot blocks:

For the compile time:

- A "2-model" compilation can usually do both: determine and process hot block properties. Generated assembly, PGO and Pronto repository can be concurrently processed with a script to extract the instruction level parallelism (ILP) information per hot block

[7]Static Single Assignment for Trees [18]: new GCC 4.x optimizer: `http://gcc.gnu.org/projects/tree-ssa/\#intro`

- Extend the code's debug-info into information containing ILP of basic blocks during compilation. It is an estimated value, not based on run-time performance. The scheduler's compile-time data could be passed through an executable itself as a triple (start block address, end block address, parallelism data). Pintool has an extensive set of APIs that accesses debug info.

For the run-time:

- Instrument the binary with a pintool that traces loops with a large number of counts (a potential knob). Then, count the number of floating point, memory and integer operations in a loop.

- PGO and Pronto may also contain ILP related ratios, which are derived from basic sampling during profiling run (with PAPI interface). Extending the Pronto repository to carry parallelism info can improve the ability of pintool's instrumentation analysis.

Additional run-time instrumentation can be based on the performance profile of the hot computational intensive blocks by running sampling along with instrumentation.[8]

This schema shows the feasibility of obtaining a process property. However, possible performance overhead of sampling and processing (even if it is incorporated in one instrumentation-sampling step) may be too heavy to make run-time decisions for the OS scheduler.

Now we will analyze the data collection process for the performance profile-aware OS scheduler. First, let's exclude 2-step models as inappropriate schemas for OS scheduling. Assume the OS cannot contain low overhead continuous sampling, then, a pintool instrumentation embedded into a running process cannot be extensive but can be discrete.

We will also assume that estimated ILP information and compiler scheduler's data can be retrieved via debug information for each basic block[9]. Instrumentation can count frequency and count of each basic block determining the estimated heat of the block.

On the run-time side, we would require implementation of one of the following: limited sampling, processing of Pronto repository, or decomposing compiler scheduler decision for the length of one basic block. We propose that context switch time might be an appropriate place to insert this lightweight process.

Pin instrumentation can be done on a basic block granularity with Pin itself setting up instrumentation calls, which is greatly improves performance.

Thus, we are considering three approaches for dynamic performance adaptive scheduler:

1. Annotation, no instrumentation. A limited lightweight instrumentation is done only on the level of basic blocks. This instrumentation would not be based on performance counters, clock cycles or actual ILP info. This is assuming some basic compiler scheduling data can be incorporated in to an executable using mechanisms similar to debug symbols. This would provide an estimated code block profile. As soon as a code block gets to a specified heat (user pre-defined knob on loop iteration count), the pintool triggers an internal OS scheduling event carrying the code profile signature.

2. Limited sampling. As noted earlier, limited sampling may be possible at the OS scheduler's

[8]The unique performance profile reflecting compute intensive block properties would not be disrupted by instrumentation performance overhead

[9]The Pronto data is mapped using debug info in DWARF2 format. Some compiler-based info such as predicted ratios IPC or FPU/ALU/SIMD utilization could be added to pronto repository data derived from pre-characterized hot blocks

checkpoint, such as context switch. This could refine information obtained from item 1 above and give more accurate data on actual ILP. A single sampling iteration over basic block could detect performance profile hazards based on counters which are specific to compute intensive blocks shown above. A trial sampling run would last only during the length of a single iteration of the loop, assuming a context switch had occurred several times during execution of a large loop count.

3. Instrumentation under "2-compile" model. Assuming ILP information can be incorporated into the binary, we would use PGO or Pronto mechanisms to generate actual sampling ratios within the profile feedback repository. After a training run we collect the profile information which includes each basic block's frequency and count, along with its ILP info. Assuming this information is available in the binary, this would indicate to the OS scheduler the performance properties of the running process. This workflow would be enabled by a simple pintool instrumentation that is analyzing each basic block's information.

From the workflow above, there is an obvious conclusion that the loader/linker has to have certain abilities to map and maintain new information passed within the generated binary. Investigating the glibc code on potential changes for loader/linker in *elf/dl-open.c*, *dl-sym.c* and *dl-load.c,* we noted a possibility of creating a number of loading threads that could load libraries in parallel. With *_dl_map_object_from_fd()* each of the threads would retrieve various information carried in the executable by link-time procedure of locating symbols. In this way hashing mechanism for objects with large amount of symbols can be parallelized in *dl_lookup_symbol_x()*, calling the expensive hashing algorithm *do_lookup_x()*. However, conducting this experiment any further is out of scope for this paper.

It is appropriate to comment on describing possible workflow combinations of "sampling-analysis" of the static algorithm for the OS performance adaptive scheduler. Due to the nature of the usage model, the static algorithm may be suitable for feasibility study or prototyping an approach, but least likely used in real life and therefore is not mentioned in this paper in detail.

Each of the workflows discussed require certain capabilities to be developed:

1. Sampling drivers are closed source but can be distributed. The open source interface for TB5 format analysis should be implemented by PAPI or VTune Analyzer/SEP.

Most of the performance events required for determining the code's performance profile are public.

2. Compiler (GCC). The performance analysis tool with basic profile feedback and vectorizer reports mechanisms already exist. The following enhancements would be needed:

- The vectorizer reports must include compiler scheduling information on parallelism.

- Mechanisms to incorporate compiler reports per basic block in to a binary need to be developed.

- A utility which collects sampling data for profile feedback needs to be developed based on the PAPI interface.

3. Pronto repository and profrun utility. These utilities currently exist as a part of Intel Compiler, but are closed source because they use the VTune TB5 file format. The following enhancements would be needed:

- PAPI interface for profrun utility and Pronto repository

- Pronto using pintool instrumentation

4. Pin. The following enhancements would be needed:

- API extensions to retrieve compiler scheduler info that are embedded into a binary

- Compiler scheduler decomposition API (an APIs that retrieve compiler's scheduler information, especially related to ILP)

- API ability to read Pronto repository from memory or a file

- A 'timer' pintool to help development activities to track performance (via gettime())

- New pintool instrumentation libraries to provide description of hazardous performance event sequences based on common code and performance profiles

- New Pin APIs that can perform independent sampling via PAPI interface to hide architecture dependences, "A Pintool doing Pronto"

- For each pintool instrumentation to specify a set of performance counters that may be affected by instrumentation itself.[10]

5. Loader/linker. The loader can be easily instrumented with the Pin interface relying on IMG_ API set. Following enhancements would be needed:

- Properly dispatch additional compiler's scheduler information embedded into binary, similarly to the debug info

- For faster linking, improve the OS loader's speed by creating loading threads

[10]These performance counters or ratios may not be present on your architecture with specified name but on modern architectures assumed to have similar ones

6. VTune analyzer. Following enhancements would be filed to Intel VTune development team:

- Ability to recognize and sample pintool instrumented code.

- Capability to receive a signal from instrumented code in order to display and translate process context and stack frame.

7. Debugger (GDB):

- Compile-time feedback: Enable reading basic block ILP information along with debug information incorporated by the compiler's scheduler

- Run-time feedback: Enable reading Pronto repository with (frequency, count) information of the BBL. It may eliminate the need for pintool instrumentation for the debugger.

- Consider scripting language to describe event sequence and pintool instrumentation algorithms.

8. OS Scheduler:

- Need to have the ability to retrieve process profile signature which characterizes performance and code constructs derived from running executable.

In order to test the feasibility of the suggested tools without changes in the kernel, we can write emulation application with the user mode simplified scheduler's algorithm by setting affinity with process's profile data.

- Have the option of signaling to the pintool instrumentation process that a context switch is about to occur and start a lightweight instrumentation mechanism with non-intrusive sampling for one iteration of a basic block. Pintool may communicate with the OS scheduler via ioctl, considering the events are coming from a driver.

3 Conclusion

In this paper we surveyed ideas spanning several technologies and tools developed by a vast community during past 10 years. We tried to bridge recent accomplishments in mainstream compiler technology, performance counters, pattern recognition algorithms, advanced binary instrumentation tools, debugging approaches and advanced dynamic optimization techniques. Many of these technologies were also inherited from previous researches on databases optimizations and compression algorithms. Demonstrated complex workflows incorporating "sample-analyze" technologies into enhanced run-time Linux infrastructure make another step towards advanced dynamic optimizations, debugging and process scheduling. Quantifying the significance and usefulness of the proposed approaches is a subject of a separate research and experiments.

Please refer to Appendix B for potential applications of the suggested ideas.

4 Acknowledgements

I would like to thank my colleagues and highlight their involvement in supplying important supporting materials, thoughts and comments:

Siddha Suresh, Venkatesh Pallipadi: Intel, Open Source Technology Center

John Piper, Anton Chernoff, Shalom Goldenberg: Intel, Dynamic Optimization Labs;

Robert Cohn, Harish Patil: Intel

Raymond Paik: Intel, Channel Product Group

5 Appendix A. Technology Background and Current Situation

Most modern micro-processors have a PMU—virtual or physical performance monitoring unit that contains hundreds or even thousands of performance counters and events. Modern micro-architectural profiling technology is divided into two distinct steps: sampling and analysis. The sampling mechanism records instruction pointers (IP) with performance counters as in (IP, frequency, count) or (IP, value). The analysis processes sampling data on the maximum time spent in repeated blocks (hot blocks), possibly including: disassembly, mapping to the source code, affiliating to a function, a process or a thread;

As the building blocks of the proposed workflow, it is important to overview existing tools and technology. Some of these tools are Open Source, some are proprietary or have a closed source engine with a BSD-style license for free distribution.

Sampling tools:

Well-known profiling tools and programming interfaces (such as VTune analyzer, EMON, PAPI, Compiler's profile guided optimization (PGO) with sampling) are usually system-wide and process agnostic. Sampling tools can be attached to any running process, but do not have access to full run-time environment and functionality context: thread storage, register values, loop count, frequency and stack.

Another type of sampling tool does not have the concept of time and is built upon executed instructions along the execution path. Such tools are not aware of stalls and clock cycles, but can sample executed instruction properties such as instruction count, instruction operands, branches, addresses, functions, images, etc.

Pin [8], [17] can be considered as a "JIT" (just-in-time) compiler, with the originating binary execution intercepted at a specified granularity. This execution is almost identical to the original. Pin contains examples of instrumentation tools like basic block profilers, cache simulators, instruction trace generators, etc. It is easy to derive new Pin-based tools using the examples as a template.

Analysis tools:

Well known analysis tools are compilers and debuggers. Beyond actual code generation and instruction scheduling, the compiler has the ability to report on optimizations, scheduler heuristics and decisions, and predicted performance counter ratios (please refer to [5]). The compiler can determine code profile and estimate performance profile of any code block, specifying execution balance across CPU units.

Some sampling tools such as VTune and EMON incorporate data analysis within them.

There are advanced parts of a compiler's optimizer which can be standalone tools that are able to parse sampling files and extract profile data.

Before moving onto another class of tools, there are also more sophisticated data analysis tools which work in formal language environments. As *Sequitur* originated from compression algorithms, it is capable of defining a sequence of events as a grammar and trace event samples on correct expression composition from given sequences of samples. The Sequitur algorithm description can be found in [9]. The implementation of sequitur has public versions.

Hybrid tools:

The series of following tools are a hybrid between sampling and analysis. Most require a complex build model with up to 3-compilation model process, (see Figure 5).

Profile-guided optimization (PGO for Intel Compilers or Profile feedback for GCC) is a part of many modern compilers [3], [4]. Currently, the PGO mostly samples executed branches, calls, frequency and loop counts with following output of the data in an intermediate format to the disk. The PGO analysis is also a part of compiler's optimizer, which is invoked with second compilation. It assists the compiler's scheduler to make better decisions by using actual run-time data instead of heuristics targeting probable application behavior.

As an extension of the PGO mechanism, a tool incorporating a trace of run-time specific events and samples (for instance, actual memory latency, cache misses) has been developed. This mechanism can be considered as a bundled sampling tool of *profrun utility* with analysis tool *pronto_tool* [6], [16], which we will refer to as Pronto. Rather then just being a natural extension of PGO capabilities these tools are standalone and not incorporated into the compiler. Architecture-wise, profrun is built upon proprietary sampling drivers spilling the data on the disk, which is called *Pronto repository*. Profrun is currently incorporated in the Intel Compiler package using Intel sampling drivers, but conceptually can be based on open source *PAPI interface*, see [7] for PAPI documentation. The pronto_tool reads and analyzes the Pronto repository for various data representation. A typical output is shown in Figure 4.

A hybrid tool *Pintools*, based on Pin, is a critical component of this paper's focus. Pintools incorporate into a single executable targeted instrumentation and analysis. This is an implementation powered by Pin API callbacks providing instrumentation for any running image at any granularity. Pintools mechanism can be considered a generic binary instrumentation template to create your own hybrid of sampling

```
$ profrun -dcache mark
$ pronto_tool -d 10 pgopti.hpi

PRONTO: Profiling module "mark":
PRONTO: Reading samples from TB5 file 'pgopti.tb5'
PRONTO: Reading samples for module at path: 'mark'

Dumping PRONTO Repository

Sample source 0: pgopti.tb5 UID: TYPE = TB5SAMP (54423553 414d5000
80ac9d3c 8f39c501 0043363d 8f39c501 00000000 00000000)

Module: "mark"
Event: "DCache miss": 35 samples
 #0  :  1 samples: [0x00001c70] mark.c:main(20:14)
        total latency=17       maximum latency=      17
        [0:7]=0    [8:15]=0    [16:31]=1  [32:99]=0  [100:inf]=0
 #1  :  5757 samples: [0x00001701] mark.c:main(23:21)
        total latency=   43132 maximum latency=     366
        [0:7]=4070 [8:15]=1668 [16:31]=18 [32:99]=0  [100:inf]=1
 #29 :  5786 samples: [0x00001700] mark.c:main(23:42)
        total latency=   40047 maximum latency=     439
        [0:7]=5294  [8:15]=433  [16:31]=55  [32:99]=0  [100:inf]=4
```

Figure 4: `pronto_tool` output

and analysis implementations. Current Pin instrumentation capabilities can extract only profiles that are not related to actual clock cycles. For example, taken branches, loop iterations counts, calls, memory references, etc.

Other examples of more sophisticated pintools-based technologies are helper threads [10.2] and hot stream data prefetch [2].

Helper thread technology, or software-based speculative pre-computation (SSP) was originated from complex database architectures and based on compiler-based pre-execution [10.1] to generate a thread that would prefetch long latency memory accesses in runtime. This is the 3-compilation model static technique. Its implementation is currently done in Intel Compilers [16] with Pronto mechanisms (option used for the first compilation

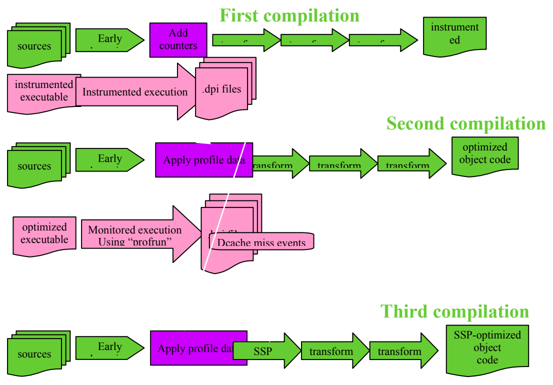


Figure 5: Helper Threads (SSP) build diagram

`--prof-gen-sampling`, for the second `--prof-use --ssp`, and for the third with `--ssp`). The workflow diagram is shown in Figure 5.

As a dynamic equivalent to this technique, the

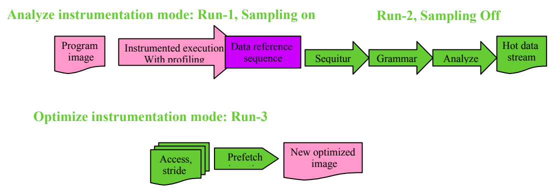


Figure 6: Hot data stream prefetch injection algorithm

mechanism described in [7] incorporates Pintool for dynamic instrumentation of memory read bursts in the sampling mode; the Sequitur for fast dynamic analysis of memory access sequences with long latencies; and matching mechanism for sequences that would benefit by prefetching (called hot data streams detection phase). Based on hot data streams, the Pin can inject prefetching instructions as shown in Figure 6.

6 Appendix B. Usage Model Examples On Proposed Workflows

Here are some potential applications of profile guided debugging and performance adaptive scheduling:

1. OS scheduling decisions may be based on occurring patterns of hardware performance events, event hazards detection or platform resource utilization hazards:

   Some event sequences can determine a hazard, upon which the OS scheduler may redefine priorities in the run queue and affinity to a logical/physical CPU.

2. Hyper-threading and Dual Core. Immediate performance gains. If a recurring pattern of utilization similar CPU resources was detected, the thread affinity assigned should distribute to run these threads on different physical cores. This approach expected to show immediate performance gains on HT-enabled system on a series of dedicated applications.

3. Independence of usage model while adopting Dual-Core/Multi-Core. In order to adopt DC/MC for maximizing the system performance, a user should be aware of system usage model. With performance adaptive scheduling infrastructure, the usage model alternation will become less relevant for performance. In turn, it may stimulate efficient adoption of multicore technology by application developers, since user awareness of usage model will not affect extracting optimal performance from the software.

4. Simplify software development schemes. Background/foreground and process priority management based on performance
   .

5. Hybrid of OpenMP & MPI for high performance programming will be simplified. A performance-adaptive OS Scheduler will handle optimal scheduling dependent on processor's resource utilization for each OpenMP thread.

6. Power utilization optimization and energy control. Modern micro-architectures have an extensive set of energy control related performance counters. When power restrictions are enforced for a process execution, the number of stall cycles due to platform resource saturation should be minimized. The optimized scheduling for processes on preventing such platform performance hazards to occur should be handled by OS scheduler.

7. Dynamic Capacity planning analysis. Analyzing profiling data logs per thread and

detection of certain event sequence hazards may assist in identifying capacity requirements for the application.

8. Out-of-order execution layer for statically scheduled codes, better utilization of "free" CPU cycles and compensation for possible compiler's scheduler inefficiencies.

   Having performance feedback based OS scheduler will provide information with additional granularity (on top of the compiler scheduler) for filling the empty cycles generated by the compiler (or if present, even during OOO execution on x86).

9. Virtualization Technology. When a code is running on virtual processing units, and utilizing a virtual pool of resources, it is important to provide optimal performance, a dynamic code migration suggestion. The assignment between virtual and physical processing unit should be done based on actual performance execution statistics. If Linux is a "guest" OS, the presence of performance adaptive scheduling mechanism will allow the OS scheduler to be aware of resource utilization across all the virtual processes.

10. Profile-guided debugging proposal targets most difficult areas of debugging - performance debugging and scalability issues. See [10].6 on examples on how to utilize Helper Threads technology for memory debugging. By combining principles of Profile-Guided optimization and conventional debugging mechanisms we showed it is possible to architect a debugger's extension to set a breakpoint at a performance or power pattern occurrence. As a result, variety of metrics for the performance may be reflected in the debugging, such as: ratio mips/watt, instruction level parallelism. State-of-the-art debuggers allow users to manually define a breakpoint on expressions which involve values obtained from the memory during the application execution. This approach assists to extend the mechanisms to combine the expression values received from CPU and chipset performance counters in run-time.

    Examples of possible breakpoints which would be set by a user who debugs multi-threaded applications are:

    - Hazardous spin locks
    - Shared memory race conditions
    - Too long or too short object waits
    - Heavy ITLB, Instruction or Trace Cache misses
    - Power consuming blocks; Floating point intensive procedures
    - Loops with extremely low CPIs; low power blocks
    - Long latency memory bus operations
    - Irregular data structure accesses; alignment issues during run-time
    - Queue and pipeline flushes, unexpected long latency execution
    - Opcode or series of opcodes being executed
    - Hyper-threading contentions or race conditions
    - OpenMP issues

There are already working applications with Pin-based instrumentation for simulation and performance prediction purposes. Extending these simulation technologies [15], similar to the PGD technique generating *Interrupt 3*, we would be able to emit other interrupts, signals and eventually generate an alternate sequence of events.

References

[1] "Enhancements for Hyper-Threading Technology in the Operating Systems – Seeking the Optimal Scheduling", by Jun Nakajima and Venkatesh Pallipadi, Intel Corporation

[2] "Dynamic Hot Data Stream Prefetching for General-Purpose Programs", by Trishul M.Chilimbi and Martin Hirzel, Microsoft Research and University of Coloroado

[3] Compiler for PGO (profile feedback) and its repository data coverage: `http://gcc.gnu.org/onlinedocs/gccint/Profile-information.htm`

[4] Compiler optimizer, gcc4.1: `http://gcc.gnu.org/onlinedocs/gcc-4.1.0/gcc/Optimize-Options.html`

[5] Compiler for vectorization and reports: `http://www.gnu.org/software/gcc/projects/tree-ssa/vectorization.html`

[6] Pronto repository content, Profrun and pronto_tool – Intel Compiler tools, based on 2-compile model: `http://www.intel.com/software/products/compilers/clin/docs/main\_cls/index.htm`

[7] PAPI: `http://icl.cs.utk.edu/papi/overview/index.html`

[8] Pin & pintool: `http://rogue.colorado.edu/pin`;
Pin manual for x86: `http://rogue.colorado.edu/pin/documentation.php`;
Pin related papers: `http://rogue.colorado.edu/pin/papers.html`

[9] **Sequitur: For Sequitur Algorithm description see**:

[9.1] "Compression and explanation in hierarchical grammars", by Craig G. Nevill-Manning and Ian H. Witten, University of Waikato, New Zealand

[9.2] "Identifying Hierarchical Structure in Sequences: A linear time algorithm", Craig G.Nevill-Manning and Ian H.Witten, University of Waikato, New Zealand, 1997

[9.3] "Efficient Representation and Abstractions for Quantifying and Exploiting Data Reference Locality", by Trishul M.Chilimbi, Microsoft Research, 2001

[10] **Helper threads and compiler based pre-execution:**

[10.1] For concept overview see "Compiler Based Pre-execution", Dongkeun Kim dissertation, University of Maryland, 2004,

[10.2] Threads: Basic Theory and Libraries: `http://www.cs.cf.ac.uk/Dave/C/node29.html`

[10.3] Usage model for helper threads in "Helper threads via Multi-threading", IEEE Micro, 11/2004

[10.4] "Helper Threads via Virtual Multithreading on an experimental Itanium 2 Processor-based platform", by Perry Wang *et al*, Intel, 2002

Helper threads and pre-execution technology for:

[10.5] Profiling, see "Profiling with Helper threads", T.Tokunaga and T. Sato (Japan), 2006

[10.6] Debugging, see "HeapMon: A helper thread approach to programmable, automatic, and low overhead memory bug detection", IBM Journal of Research and Development, by R.Shetty et al., 2005

[11] "Dynamic run-time architecture technique for enabling continuous optimizations" by Tipp Moseley, Daniel A. Connors, etc., University of Colorado

[12] "Chip Multithreading Systems Need a New Operating System Scheduler" by Alexandra Fedorova, Christopher Small, et all, Harward University & Sun Micro.

[13] "Methods for Modeling Resource Contention on Simultaneous Multithreading Processors" by Tipp Moseley, Daniel A. Connors, University of Colorado

[14] "Pthreads Primer, A guide to multithreaded programming", Bill Lewis and Daniel J. Berg, SunSoft Press, 1996

[15] SimPoint toolkit by UCSD: `http://www-cse.ucsd.edu/~calder/simpoint/simpoint_overview.htm`

[16] Intel Compiler Documentation – keywords: Software-based Speculative Precomputation (SSP); Profrun utility, prof-gen-sampling: `http://www.intel.com/software/products/compilers/clin/docs/main_cls/index.htm`

[17] "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," by Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, Kim Hazelwood. Programming Language Design and Implementation (PLDI), Chicago, IL, June 2005

[18] Tree SSA: A new optimization infrastructure for GCC, by Diego Novillo, Red Hat Canada, 2003: `http://people.redhat.com/dnovillo/pub/tree-ssa/papers/tree-ssa-gccs03.pdf`; `http://gcc.gnu.org/projects/tree-ssa/#intro`

A Reliable and Portable Multimedia File System

Joo-Young Hwang, Jae-Kyoung Bae, Alexander Kirnasov,
Min-Sung Jang, Ha-Yeong Kim
Samsung Electronics, Suwon, Korea
{jooyoung.hwang, jaekyoung.bae, a78.kirnasov}@samsung.com
{minsung.jang, hayeong.kim}@samsung.com

Abstract

In this paper we describe design and implementation of a database-assisted multimedia file system, named as XPRESS (eXtendible Portable Reliable Embedded Storage System). In XPRESS, conventional file system metadata like inodes, directories, and free space information are handled by transactional database, which guarantees metadata consistency against various kinds of system failures. File system implementation and upgrade are made easy because metadata scheme can be changed by modifying database schema. Moreover, using well-defined database transaction programming interface, complex transactions like non-linear editing operations are developed easily. Since XPRESS runs in user level, it is portable to various OSes. XPRESS shows streaming performance competitive to Linux XFS real-time extension on Linux 2.6.12, which indicates the file system architecture can provide performance, maintainability, and reliability altogether.

1 Introduction

Previously consumer electronics (CE) devices didn't use disk drives, but these days disks are being used for various CE devices from personal video recorder (PVR) to hand held camcorders, portable media players, and mobile phones. File systems for such devices have requirements for multimedia extensions, reliability, portability and maintainability.

Multimedia Extension Multimedia extensions required for CE devices are non-linear editing and advanced file indexing. As CE devices are being capable of capturing and storing A/V data, consumers want to personalize media data according to their preference. They make their own titles and shares with their friends via internet. PVR users want to edit recorded streams to remove advertisement and uninterested portions. Non-linear editing system had been only necessary in the studio to produce broadcast contents but it will be necessary also for consumers. Multimedia file system for CE devices should support non-linear editing operations efficiently. File system should support file indexing by content-aware attributes, for example the director and actor/actress of a movie clip.

Reliability On occurrence of system failures(*e.g.* power failures, reset, and bad blocks), file system for CE devices should be recovered to a consistent state. Implementation of a reliable file system from scratch or API extension to existing file system are difficult and requires long stabilization effort. In case of appending multimedia API extension (*e.g.* non-linear edit operations), reliability can be a main concern.

Portability Conventional file systems have dependency on underlying operating system. Since CE devices manufacturers usually use various operating systems, file system should be easily ported to various OSes.

Maintainability Consumer devices are diverse and the requirements for file system are slightly different from device to device. Moreover, file system requirements are time-varying. Maintainability is desired to reduce cost of file system upgrade and customization.

In this paper, we propose a multi-media file system architecture to provide all the above requirements. A research prototype named as "XPRESS"(eXtensible Portable Reliable Embedded Storage System) is implemented on Linux. XPRESS is a user-level database-assisted file system. It uses a transactional Berkeley database as XPRESS metadata store. XPRESS supports zero-copy non-linear edit (cut & paste) operations. XPRESS shows performance in terms of bandwidth and IO latencies which is competitive to Linux XFS. This paper is organized as follows. Architecture of XPRESS is described in section 2. XPRESS's database schema is presented in section 3. Space allocation is detailed in section 4. Section 5 gives experimental results and discussions. Related works are described in section 6. We conclude and indicate future works in section 7.

2 Architecture

File system consistency is one of important file system design issue because it affects overall design complexity. File system consistency can be classified into metadata consistency and data consistency. Metadata consistency is to support transactional metadata operations with ACID (atomicity, consistency, isolation, durability) semantics or a subset of ACID. Typical file system metadata consist of inodes, directories, disk's free space information, and free inodes information. Data consistency means supporting ACID semantics for data transactions as well. If a data transaction to update a portion of file is aborted due to some reasons, the data update is complete or data update is discarded at all.

There have been several approaches of implementing file system consistency; log structured file system [11] or journaling file system [2]. In log structured file system, all operations are logged to disk drive. File system is structured as logs of consequent file system update operations. Journaling is to store operations on separate journal space before updating the file system. Journaling file system writes twice (to journal space and file system) while log structured file system does write once. However journaling approach is popular because it can upgrade existing non-journaling file system to journaling file system without losing or rewriting the existing contents.

Implementation of log structured file system or journaling is a time consuming task. There have been a lot of researches and implementation of ACID transactions mechanism in database technology. There are many stable open source databases or commercial databases which provide transaction mechanism. So we decide to exploit databases' transaction mechanism in building our file system. Since we aimed to design a file system with performance and reliability altogether, we decided not to save file contents in database. Streaming performance can be bottlenecked by database if file contents are stored in db. So, only metadata is stored in database while file contents are stored to a partition. Placement of file contents on the data partition is guided by multimedia optimized allocation strategy.

Storing file system metadata in database makes file system upgrades and customization much easier than conventional file systems. XPRESS handles metadata thru database API and is not responsible for disk layout of the metadata. File system developer has only to design high level database schema and use well defined database API to upgrade existing database. To upgrade conventional file systems, developer should handle details of physical layout of metadata and modify custom data structures.

XPRESS is implemented in user level because user level implementation gives high portability and maintainability. File system source codes are not dependent on OS. Kernel level file systems should comply with OS specific infrastructures. Linux kernel level file system compliant to VFS layer cannot be ported easily to different RTOSes. There can be a overhead which is due to user level implementation. XPRESS should make system calls to read/write block device file to access file contents. If file data is sparsely distributed, context switching happens for each extent. There was an approach to port existing user level database to kernel level[9] and develop a kernel level database file system. It can be improved if using Linux AIO efficiently. Current XPRESS does not use Linux AIO but has no significant performance overhead.

Figure 1 gives a block diagram of XPRESS file system. XPRESS is designed to be independent of database. DB Abstraction Layer (DBAL) is located between metadata manager and Berkeley DB. DBAL defines basic set of interfaces which modern databases usually have. SQL or complex data types are not used. XPRESS has not much OS dependencies. OSAL of XPRESS has only wrappers for interfacing block device file which may be different across operating systems.

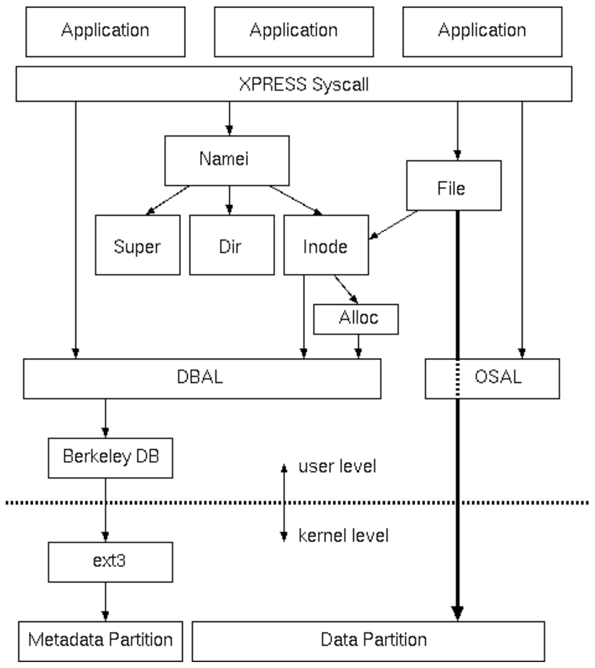


Figure 1: Block Diagram of XPRESS File System

There are four modules handling file system metadata; superblock, directory, inode, and alloc modules. Directory module implements the name space using a lookup database (`dir.db`) and `path_lookup` function. The function looks up the path of a file or a directory through recursive DB query for each token of the path divided separated by '/' character. The flowing is the simple example for the process of this function. Superblock module maintains free inodes. Inode module maintains the logical-to-physical space mappings for files. Alloc module maintains free space.

In terms of file IO, file module calls inode module to updates or queries `extents.db` and reads or writes file contents. The block device file corresponding to the data partition (say "`/dev/sda1`") is opened for read/write mode at mount time and its file descriptor is saved in environment descriptor, which is referred to by every application accessing the partition. XPRESS does not implement caching but relies on Linux buffer cache. XPRESS supports both direct IO and cached IO. For cached IO, the block device file is opened without `O_DIRECT` flag while opened with `O_DIRECT` in case of direct IO.

Support for using multiple partitions concurrently, XPRESS manages mounted partitions information using environment descriptors. A environment descriptor has information about a partition and its mount point, which is used for name space resolution. Since all DB resources (transaction, locking, logging, etc) corresponding to a partition belongs to a DB environment and separate logging daemon is ncessary for separate environment, a new logging daemon is launched on mounting a new partition.

3 Transactional Metadata Management

3.1 Choosing Database

As the infrastructure of XPRESS, database should conform to the following requirements. First, it should have transactional operation and recovery support. It is important for implementing metadata consistency of XPRESS. Second, it should be highly concurrent. Since file system is used by many threads or processes, database should support and have high concurrency performance as well. Third, it should be light-weight. Database does not have to support many features which are unnecessary for file system development. It only has to provide API necessary for XPRESS efficiently. Finally, it should be highly efficient. Database implemented as a separate process has cleaner interface and maintainability; however library architecture is more efficient.

Berkeley DB is an open source embedded database that provides scalable, high-performance, transaction-protected data management services to applications. Berkeley DB provides a simple function-call API for data access and management. Berkeley DB is embedded in its application. It is linked directly into the application and runs in the same address space as the application. As a result, no inter-process communication, either over the network or between processes on the same machine, is required for database operations. All database operations happen inside the library. Multiple processes, or multiple threads in a single process, can all use the database at the same time as each uses the Berkeley DB library. Low-level services like locking, transaction logging, shared buffer management, memory management, and so on are all handled transparently by the library.

Berkeley DB offers important data management services, including concurrency, transactions, and recovery. All of these services work on all of the storage structures. The library provides strict ACID transaction semantics, by default. However, applications can relax the isolation or the durability. XPRESS uses relaxed semantics for performance. Multiple operations can be grouped into a single transaction, and can be committed or rolled back atomically. Berkeley DB uses a technique called two-phase locking to support highly concurrent transactions, and a technique called write-ahead logging to guarantee that committed changes survive application, system, or hardware failures.

If a BDB environment is opened with a recovery option, Berkeley DB runs recovery for all databases belonging to the environment. Recovery restores the database to a clean state, with all committed changes present, even after a crash. The database is guaranteed to be consistent and all committed changes are guaranteed to be present when recovery completes.

DB	unit	key	data	structure	secondary index
super	partition	partition number	superblock data	RECNO	-
dir	partition	parent INO/file name	INO	B-tree	INO (B tree)
inode	partition	INO	inode data	RECNO	-
free space	partition	PBN	length	B-tree	length(B tree)
extents	file	file offset	PBN/length	B-tree	-

Table 1: Database Schema, INO: inode number, RECNO: record number, PBN: physical block number

3.2 Database Schema Design

XPRESS defines five databases to store file system metadata and their schema is shown in Table 1. Each B-tree database has a specific key-comparison function that determines the order in which keys are stored and retrieved. Secondary index is used for performance acceleration.

Superblock DB The superblock database, `super.db`, stores file system status and inode bitmap information. As overall file system status can be stored in just one record, and inode bitmap also needs just a few records, this database has RECNO structure, and does not need any secondary index. Super database keeps a candidate inode number, and when a new file is created, XPRESS uses this inode number and then replaces the candidate inode number with another one selected after scanning inode bitmap records.

Directory DB The dir database, `dir.db`, maps directory and file name information to inode numbers. The key used is a structure with two values: the parent inode number and the child file name. This is similar to a standard UNIX directory that maps names to inodes. A directory in XPRESS is a simple file with a special mode bit. As XPRESS is user-level file system, it does not depend on Linux VFS layer. As a result it cannot use directory related caches of Linux kernel (*i.e.*, dentry cache), instead, database cache will be used for that purpose.

Inode DB The inode database, `inode.db`, maps inode numbers to the file information (*e.g.*, file size, last modification time, etc.). When a file is created, a new inode record is inserted into this database, and when a file is deleted, the inode record is removed from this database. XPRESS assigns inode numbers in an increasing order and upper limit on the number of inodes is determined when creating the file system. It will be possible to make secondary indices for `inode.db` for efficiency (*e.g.*, searching files whose size is larger than 1M). But currently no secondary index is used.

Free Space DB The free-space database, freespace.db, manages free extents of the partition. Initially free-space database has one record whose data is just one big extent which means a whole partition. When a change in file size happens this database will update its records.

Extents DB A file in XPRESS consists of several extents of which size is not fixed. The extents database, `extents.db`, maps file offset to physical blocks address of the extent including the file data. As this database corresponds to each file, its life-time is also same with that of a file. The exact database name is identified with an inode number; `extents.db` is just database file name. This database is only dynamically removable while all other databases are going on with the file system.

3.3 Transactional System Calls

A transaction is atomic if it ensures that all the updates in a transaction are done altogether or none of them is done at all. After a transaction is ether committed or aborted, all the databases for file system metadata are in consistent. In multi-process or multi-thread environment, concurrent operation should be possible without any interference of other operations. We can say this property isolated. An Operation will have already been committed to the databases or is in the transaction log safely if the transaction is committed. So the file system operations are durable which means file system metadata is never lost.

Each file system call in XPRESS is protected by a transaction. Since XPRESS system calls are using the transactional mechanism provided by Berkeley DB, ACID properties are enabled for each file system call of XPRESS. A file system call usually involves multiple reads and updates to metadata. An error in the middle of system call can cause problems to the file system. By storing file system metadata in the databases and enabling transactional operations for accessing those databases, file system is kept stable state in spite of many unexpected errors.

An error in updating any of the databases during a system call will cause the system call, which is protected by a transaction, to be aborted. There can be no partially done system calls. XPRESS ensures that any system call is complete or not started at all. In this sense, a XPRESS system call is atomic.

Satisfying strict ACID properties can cause performance degradation. Especially in file system, since durability may not be strict condition, we can relax this property for better performance. XPRESS compromises durability by not syncing the log on transaction commit. Note that the policy of durability applies to all databases in an environment. Flushing takes place periodically and the cycle of flushing can be configurable. Default cycle is 10 seconds.

Every XPRESS system call handles three types of failures; power-off, deadlock, and unexpected operation failure. Those errors are handled according to cases. In case of power-off we are not able to handle immediately. After the system is rebooted, recovery procedure will be automatically started. Deadlock may occur during transaction when the collision between concurrent processes or threads happened. In this case, one winning process access database successfully and all other processes or threads are reported deadlock. They have to abort and retry their transactions. In case of unexpected operation failure, on-going transaction is aborted and system calls return error to its caller.

3.4 Non-linear Editing Support

Non-linear editing on A/V data means cutting a segment of continuous stream and inserting a segment inside a stream. This is required when users want to remove unnecessary portion of a stream; for example, after recording two hours of drama including advertisement, user wants to remove the advertisement segments from the stream. Inserting is useful when a user wants to merge several streams into one title. These needs are growing because consumers capture many short clips during traveling somewhere and want to make a title consisting of selected segments of all the clips.

Conventional file systems does not consider this requirement, so to support those operations, a portion of file should be migrated. If front end of a file is to be cut, the remaining contents of the file should be copied to a new file because the remaining contents are not block aligned. File systems does assume block aligned starting of a file. In other words, they do not assume that a file does not start in the middle of a block. Moreover, a block is not assumed to be shared by different files. This has been general assumptions about had disk file system because disk is a block based device which means space is allocated and accessed in blocks. The problem is more complicated for inserting. A file should have a hole to accommodate new contents inside it and the hole may not be block-aligned. So there can be breaches in the boundaries of the hole.

We solve those problems by using byte-precision extents allocation. XPRESS allows physical extent not aligned with disk block size. After cutting a logical extent, the physical extents corresponding to the cut logical extent can be inserted into other file. Implementation of those operations involves updating databases managing the extents information. Data copy is not required for the editing operations, only extents info is updated accordingly.

4 Extent Allocation and Inode Management

In this section, the term "block" refers to the allocation unit in XPRESS. The block size can be configurable at format time. For byte-precision extents, block size is configured to one byte.

4.1 Extent Allocation

There were several approaches for improving contiguity of file allocation. Traditionally (FAT, ext2, ext3) disc free space was handled by means of bitmaps. Bit 0 at position n of the bitmap designates, that n-th disc block is free (can be allocated for the file). This approach has several drawbacks; searching for the free space is not effective and bitmaps do not explicitly provide information on contiguous chunks of the free space.

The concept of extent was introduced in order to overcome mentioned drawbacks. An extent is a couple, consisting from block number and length in units of blocks. An extent represents a contiguous chunk of blocks on the disk. The free space can be represented by set of corresponding extents. Such approach is used for example in XFS (with exception of real-time volume). In case of XFS the set of extents is organized in two B+ trees, first sorted by starting block number and second sorted by size of the extent. Due to such organization the search for the free space becomes essentially more efficient compared to the case of using bitmaps.

One way to increase the contiguity comes from increasing the block size. Such approach is especially useful for real time file systems which deal with large video streams in combination with idea of using special volume specifically for these large files. Similar approach is utilized in XFS for real-time volume. Another way to improve file locality on the disc is preallocation. This technique can be described as an allocation of space for the file in advance before the file is being written to. Preallocation can be accomplished whether on application or on the file system level. Delayed allocation can also be used for contiguous allocation. This technique postpones an actual disc space allocation for the file, accumulating the file contents to be written in the memory buffer, thus providing better information for the allocation module.

XPRESS manages free space by using extents. Allocation algorithm uses two databases: `freespace1.db` and `freespace2.db`, which collect the information on free extents which are not occupied by any files on the file system. The `freespace1.db` orders all free extents by left end and the `freespace2.db`, which is secondary indexed db of `freespace1.db`, orders all free extents by length. Algorithm tries to allocate entire length of req_len as one extent in the neighborhood of the last_block. If it fails, then it tries to allocate maximum extent within neighborhood. If no free extent is found within neighborhood, it tries to allocate maximum free extent in the file system. After allocating a free extent, neighborhood is updated and it tries the same process to allocate the remaining, which iterate until entire requested length is allocated. Search in neighborhood is accomplished in left and right directions using two cursors on `freespace1.db`. The neighborhood size is determined heuristically proportional to the req_len.

Pre-allocation is important for multi-threaded IO applications. When multiple IO threads try to write files, the file module tries to pre-allocate extents enough to guarantee disk IO efficiency. Otherwise, disk blocks are fragmented because contiguous blocks are allocated to different files in case that multiple threads are allocating simultaneously. Pre-allocation size of XPRESS is by default 32Mbytes which may be varying according to disk IO bandwidth. On file close, unused spaces among the pre-allocated extents are returned to free space database, which is handled by FILE module of XPRESS.

4.2 Inode Management

File extents are stored in `extents.db`. Each file extent is represented by logical file offset and corresponding physical extent. Both offset and physical extent are specified with byte precision in order to provide facilities for partial truncation that is truncation of some portion of the file from a specified position of the file with byte precision.

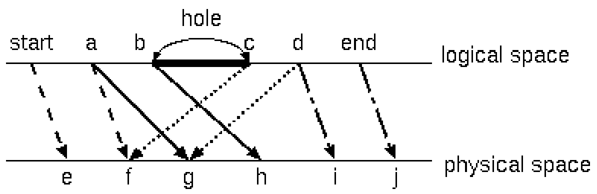


Figure 2: Logical to physical mapping

Let us designate a logical extent starting from the logical file offset a mapped to a physical extent [b,c] as [a,[b,c]] for explanations in the following. A logical file interval may contain some regions which do not have physical image on the disc; such regions are referred as holes. The main interface function of the INODE module is `xpress_make_segment_op()`. The main parameters of the function are type of the operation (READ, WRITE and DELETE) and logical file segment, specified by logical offset and the length.

`extents.db` is read on file access to convert a logical segment to a set of physical extents. Figure 2 shows an example of segment mapping. Logical segment [start, end] corresponds to following set of physical extents; {[start,[e,f]], [a,[g,h]], [b,[]], [c,[f,g]], [d,[i,k]]}, where the logical extent [b,c] is a file hole. In case of read operation to `extents.db`, the list of physical extents is retrieved and returned to the file module.

When specified operation is WRITE and specified segment contains yet unmapped area - that is writing to the hole or beyond end of file is accomplished, then allocation request may be generated after aligning the requested size to the block size since as was mentioned allocation module uses blocks as units of allocation. In Figure 3, blocks are allocated when writing the segment [start,end] of the file.

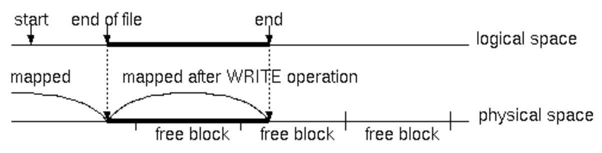
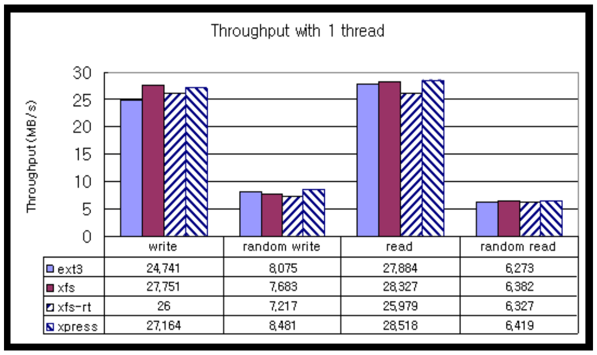


Figure 3: Block allocation details



	write	random write	read	random read
◻ ext3	24,741	8,075	27,884	6,273
■ xfs	27,751	7,683	28,327	6,382
▨ xfs-rt	26	7,217	25,979	6,327
▧ xpress	27,164	8,481	28,518	6,419

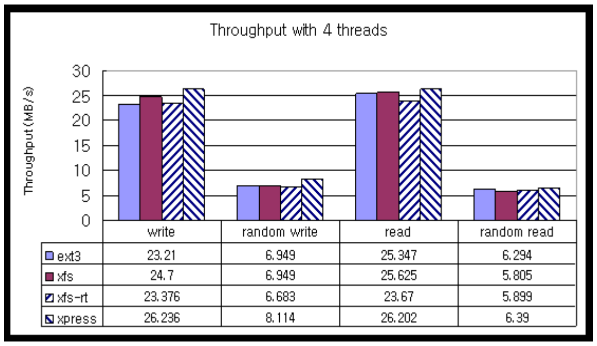
Figure 4: Throughput with 1 thread

Xpress allows to perform partial file truncate operation as well, as cut and paste operation. First operation removes some fragment of the file, while the latter also inserts the removed file portion into specified position of another file. On partial truncate operation, a truncated logical segment is left as a hole. On cut operation, logical segment mapping is updated to avoid hole. On paste operation, mapping is updated to avoid overwriting as well.

5 Experimental Results

Test platform is configured as following. Target H/W : Pentium4 CPU 1.70GHz with 128MB RAM and 30GB SAMSUNG SV3002H IDE disk
Target O/S : Linux-2.6.12
Test tools : `tiotest`(tiobench[1]), `rwrt`

XPRESS consistency semantic is metadata journaling and ordered data writing which is



	write	random write	read	random read
◻ ext3	23.21	6.949	25.347	6.294
■ xfs	24.7	6.949	25.625	5.805
▨ xfs-rt	23.376	6.683	23.67	5.899
▧ xpress	26.236	8.114	26.202	6.39

Figure 5: Throughput with 4 threads

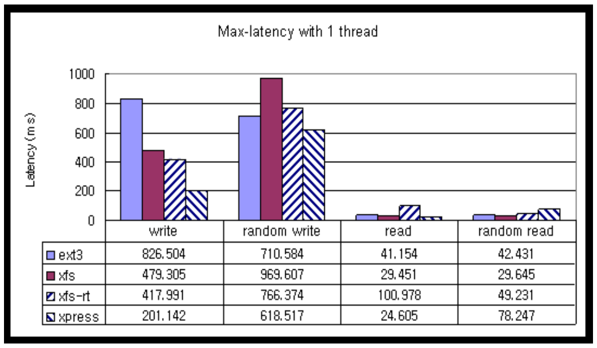
similar to XFS file system and ext3 with ordered data mode. Hence we chose XFS and ext3 as performance comparison targets. As XFS file system provides real-time sub-volume option, we also used it as one of comparison targets. By calling it XFS-RT, we will distinguish it from normal XFS. `tiotest` is threaded I/O benchmark tool which can test disk I/O throughput and latency. `rwrt` is a basic file I/O application to test ABISS (Adaptive Block IO Scheduling System)[3]. It performs isochronous read operations on a file and gathers information on the timeliness of system responses. It gives us I/O latencies or jitters of streaming files, which is useful for analyzing streaming quality. We did not use XPRESS's I/O scheduling and buffer cache module because we can get better performance with the Linux's native I/O scheduling and buffer cache.

IO Bandwidth Comparison

Figure 4 and Figure 5 show the results of I/O throughput comparison for each file system. These two tests are conducted with `tiotest` tool whose block size option is 64KB which means the unit of read and write is 64KB. In both cases, there is no big difference between all file systems.

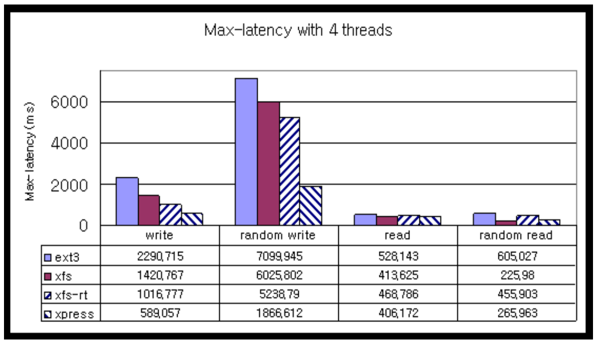
IO Latency Comparison

We used both `tiotest` and `rwrt` tool to measure I/O latencies in case of running multiple



	write	random write	read	random read
▭ ext3	826.504	710.584	41.154	42.431
■ xfs	479.305	969.607	29.451	29.645
▨ xfs-rt	417.991	766.374	100.978	49.231
▧ xpress	201.142	618.517	24.605	78.247

Figure 6: MAX latency with 1 thread



	write	random write	read	random read
▭ ext3	2290,715	7099,945	528,143	605,027
■ xfs	1420,767	6025,802	413,625	225,98
▨ xfs-rt	1016,777	5238,79	468,786	455,903
▧ xpress	589,057	1866,612	406,172	265,963

Figure 7: MAX latency with 4 threads

concurrent I/O threads. The `rwrt` is dedicated for profiling sequential read latencies and tiobench is used for profiling latencies of write, random write, read, and random read operations.

Figure 6 and Figure 7 show the maximum I/O latencies for each file system obtained from `tiotest`. The maximum I/O latency is a little low on XPRESS file system. In terms of write latency, XPRESS outperforms others while maximum read latencies are similar.

To investigate read case more, we conducted the `rwrt` with the number of threads from 1 to 8 and measure the read latencies for each read request. Each process tries its best to read blocks of a file. The results of these tests include the latencies of each read request whose size is 128Kbyte. Table 2 shows the statistics of the measured read latencies for 2, 4, and 8 threads. The average latencies are nearly the same for all experimented file systems. XPRESS show slightly smaller standard deviation than others and improvement regarding the maximum latencies. Please note that XPRESS maximum latency is 413 milliseconds while the max latency of XFS-RT is 524 milliseconds.

File System	Average	Std Dev	Max
	2 threads		
Ext3	8.47	26.03	512
XFS	8.62	25.15	239
XFS-RT	8.96	25.66	260
XPRESS	8.62	24.93	262
	4 threads		
Ext3	17.9594	72.70	877
XFS	17.8389	72.70	528
XFS-RT	18.7506	74.33	524
XPRESS	17.8467	71.79	413
	8 threads		
Ext3	36.38	166.47	1144
XFS	36.35	166.86	1049
XFS-RT	38.34	171.39	1023
XPRESS	36.33	166.38	923

Table 2: Read Latencies Statistics. All are measured in milliseconds.

Jitters during Streaming at a Constant Data Rate

The jitters performance is important from user's point of view because it leads to a low quality video streaming. We performed `rwrt` tool with ABISS I/O scheduling turned off. The `rwrt` is configured to read a stream with a constant specified data rate, say 3MB/sec or 5MB/sec. Table 3 shows jitter statistics for each file system when running single thread with 5MB/sec rate, four concurrent threads at 5MB/sec rates, and six concurrent threads at 3MB/sec rates, respectively. The results of these tests include the jitters of each read re-

File System	Average	Std Dev	Max
	1 thread (5MB/s)		
Ext3	3.97	2.44	43
XFS	3.91	2.03	46
XFS-RT	3.94	1.79	25
XPRESS	3.89	1.92	40
	4 threads (each 5MB/sec)		
Ext3	18.00	41.72	694
XFS	15.67	30.25	249
XFS-RT	19.22	34.63	267
XPRESS	17.93	38.83	257
	6 threads (each 3MB/sec)		
Ext3	24.80	48.43	791
XFS	23.81	42.20	297
XFS-RT	26.57	43.48	388
XPRESS	23.66	45.31	337

Table 3: Jitter Statistics. All are measured in milliseconds.

quest whose size is 128Kbyte. Mean values of experimented file systems are nearly the same. XPRESS, XFS, and XFS-RT show the similar standard deviation of jitters which is much less than that of ext3.

Metadata and Data Separation
Metadata access patterns are usually small requests while streaming accesses are continuous bulk data transfer. By separating metadata and data partitions on different media, performance can be optimized according to their workload types. XPRESS allows metadata to be placed in separate partition which can be placed on another disk or NAND flash memory. Table 4 summarizes the effect of changing the db partition. This is test for extreme case since we used ramdisk, which is virtual in-memory disk, as a separate disk. However, we can identify the theoretical possibility of performance enhancement from this test. According to the result, the enhancement happens mainly on write test by 11%. We expect using separate partition will reduce latencies significantly, which are not shown here.

configuration	write	read
same disk	25.20	27.82
separate disk	28.20	28.53

Table 4: Placing metadata on ramdisk and data on a disk. All are measured in MB/s.

Non-linear Editing
To test cut-and-paste operation, we prepared two files each of which is of 131072000 bytes, then cut the latter half (= 65,536,000 bytes) of one file and append it to the other file. In XPRESS, this operation takes 0.274 seconds. For comparison, the operation is implemented on ext3 by copying iteratively 65536 bytes, which took 3.9 seconds. The performance gap is due to not copying file data but updating file system metadata (`extents.db` and `inode.db`). In XPRESS, the operation is atomic transaction and can be undone if it fails during the operation. However our implementation of the operation on ext3 does not guarantee atomicity.

6 Related Works

Traditionally file system and database has been developed separately without depending on each other. They are aimed for different purposes; file system is for byte streaming and database is for advanced query support. Recently there is a growing needs to index files with attributes other than file name. For example, file system containing photos and emails need to be indexed by date, sender, or subjects. XML document provides jit(just-in-time) schema to support contents based indexing. Conventional file system is lack of indexing that kind of new types of files. There has been a few works to implement database file systems to enhance file indexing; BFS[5], GnomeStorage[10], DBFS[6], kbdbfs[9], and WinFS[7]. Those are not addressing streaming performance issues. For video streaming, data placement on disk is important. While conventional database file systems resorts to DB or other file systems for file content storage, XPRESS controls placement of files content by itself.

Compared to custom multimedia file systems (*e.g.* [12], [8], [4]), XPRESS has a well-defined file system metadata design and implementation framework and file system metadata is protected by transactional databases. Appending multimedia extensions like indexing and non-linear editing is easier. Moreover since it is implemented in user level, it is highly portable to various OSes.

7 Conclusions and Future Works

In this paper we described a novel multimedia file system architecture satisfying streaming performance, multimedia extensions (non-linear editing), reliability, portability, and maintainability. We described detailed design and issues of XPRESS which is a research prototype implementation. In XPRESS, file system metadata are managed by a transactional database, so metadata consistency is ensured by transactional DB. Upgrade and customization of file system is easy task in XPRESS because developers don't have to deal with metadata layout in disk drive. We also implement atomic non-linear editing operations using DB transactions. Cutting and pasting A/V data segments is implemented by extents database update without copying segment data. Compared to ext3, XFS, and XFS real-time sub-volume extension, XPRESS showed competitive streaming performance and more deterministic response times.

This work indicates feasibility of database-assisted multimedia file system. Based on the database and user level implementation, it makes future design change and porting easy while streaming performance is not compromised at the same time. Future works are application binary compatibility support using system call hijacking, appending contents-based extended attributes, and encryption support. Code migration to kernel level will also be helpful for embedded devices having low computing power.

References

[1] Threaded i/o tester. `http://sourceforge.net/projects/tiobench/`.

[2] M. Cao, T. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceeding of Linux Symposium*, July 2005.

[3] Giel de Nijs, Benno van den Brink, and Werner Almesberger. Active block io scheduling system. In *Proceeding of Linux Symposium*, pages 109–126, July 2005.

[4] Pallavi Galgali and Ashish Chaurasia. San file system as an infrastructure for multimedia servers. `http://www.redbooks.ibm.com/redpapers/pdfs/redp4098.pdf`.

[5] Dominic Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers, Inc., 1999. ISBN 1-55860-497-9.

[6] O. Gorter. Database file system. Technical report, University of Twente, aug 2004. `http://ozy.student.`

utwente.nl/projects/dbfs/dbfs-paper.pdf.

[7] Richard Grimes. Revolutionary file storage system lets users search and manage files based on content, 2004. http://msdn.microsoft.com/msdnmag/issues/04/01/WinFS/default.aspx.

[8] R. L. Haskin. Tiger Shark — A scalable file system for multimedia. *IBM Journal of Research and Development*, 42(2):185–197, 1998.

[9] Aditya Kashyap. *File System Extensibility and Reliability Using an in-Kernel Database*. PhD thesis, Stony Brook University, 2004. Technical Report FSL-04-06.

[10] Seth Nickell. A cognitive defense of associative interfaces for object reference, Oct 2004. http://www.gnome.org/~seth/storage/associative-interfaces.pdf.

[11] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[12] Philip Trautman and Jim Mostek. Scalability and performance in modern file systems. http://linux-xfs.sgi.com/projects/xfs/papers/xfs_white/xfs_white_paper.%html.

Utilizing IOMMUs for Virtualization in Linux and Xen

Muli Ben-Yehuda
muli@il.ibm.com

Jon Mason
jdmason@us.ibm.com

Orran Krieger
okrieg@us.ibm.com

Jimi Xenidis
jimix@watson.ibm.com

Leendert Van Doorn
leendert@us.ibm.com

Asit Mallick
asit.k.mallick@intel.com

Jun Nakajima
jun.nakajima@intel.com

Elsie Wahlig
elsie.wahlig@amd.com

Abstract

IOMMUs are hardware devices that translate device DMA addresses to proper machine physical addresses. IOMMUs have long been used for RAS (prohibiting devices from DMA'ing into the wrong memory) and for performance optimization (avoiding bounce buffers and simplifying scatter/gather). With the increasing emphasis on virtualization, IOMMUs from IBM, Intel, and AMD are being used and re-designed in new ways, e.g., to enforce isolation between multiple operating systems with direct device access. These new IOMMUs and their usage scenarios have a profound impact on some of the OS and hypervisor abstractions and implementation.

We describe the issues and design alternatives of kernel and hypervisor support for new IOMMU designs. We present the design and implementation of the changes made to Linux (some of which have already been merged into the mainline kernel) and Xen, as well as our proposed roadmap. We discuss how the interfaces and implementation can adapt to upcoming IOMMU designs and to tune performance for different workload/reliability/security scenarios. We conclude with a description of some of the key research and development challenges new IOMMUs present.

1 Introduction to IOMMUs

An I/O Memory Management Unit (IOMMU) creates one or more unique address spaces which can be used to control how a DMA operation from a device accesses memory. This functionality is not limited to translation, but can also provide a mechanism by which device accesses are isolated.

IOMMUs have long been used to address the disparity between the addressing capability of some devices and the addressing capability of the host processor. As the addressing capability of those devices was smaller than the addressing capability of the host processor, the devices could not access all of physical memory. The introduction of 64-bit processors and the Physical Address Extension (PAE) for x86, which allowed processors to address well beyond the 32-bit limits, merely exacerbated the problem.

Legacy PCI32 bridges have a 32-bit interface which limits the DMA address range to 4GB. The PCI SIG [11] came up with a non-IOMMU fix for the 4GB limitation, Dual Address Cycle (DAC). DAC-enabled systems/adapters bypass this limitation by having two 32-bit address phases on the PCI bus (thus allowing 64 bits of total addressable memory). This modification is backward compatible to allow 32-bit, Single Address Cycle (SAC) adapters to function on DAC-enabled buses. However, this does not solve the case where the addressable range of a specific adapter is limited.

An IOMMU can create a unique translated address space, that is independent of any address space instantiated by the MMU of the processor, that can map the addressable range of a device to all of system memory.

When the addressable range of a device is limited and no IOMMU exists, the device might not be able to reach all of physical memory. In this case a region of system memory that the device can address is reserved, and the device is programmed to DMA to this reserved area. The processor then copies the result to the target memory that was beyond the "reach" of the device. This method is known as *bounce buffering*.

In IOMMU isolation solves a very different problem than IOMMU translation. Isolation restricts the access of an adapter to the specific area of memory that the IOMMU allows. Without isolation, an adapter controlled by an untrusted entity (such as a virtual machine when running with a hypervisor, or a non-root user-level driver) could compromise the security or availability of the system by corrupting memory.

The IOMMU mechanism can be located on the device, the bus, the processor module, or even in the processor core. Typically it is located on the bus that bridges the processor/memory areas and the PCI bus. In this case, the IOMMU intercepts all PCI bus traffic over the bridge and translates the in- and out-bound addresses. Depending on implementation, these in- and out-bound addresses, or translation window, can be as small as a few megabytes to as large as the entire addressable memory space by the adapter (4GB for 32-bit PCI adapters). If isolation is not an issue, it may be beneficial to have addresses beyond this window pass through unmodified.

AMD IOMMUs: GART, Device Exclusion Vector, and I/O Virtualization Technology

AMD's Graphical Aperture Remapping Table (GART) is a simple translation-only hardware IOMMU [4]. GART is the integrated translation table designed for use by AGP. The single translation table is located in the processor's memory controller and acts as an IOMMU for PCI. GART works by specifying a physical memory window and list of pages to be translated inside that window. Addresses outside the window are not translated. GART exists in the AMD Opteron, AMD Athlon 64, and AMD Turion 64 processors.

AMD's Virtualization (AMD-V(TM) / SVM) enabled processors have a Device Exclusion Vector (DEV) table define the bounds of a set of protection domains providing isolation. DEV is a bit-vectored protection table that assigns per-page access rights to devices in that domain. DEV forces a permission check of all device DMAs indicating whether devices in that domain are allowed to access the corresponding physical page. DEV uses one bit per physical 4K page to represent each page in the machine's physical memory. A table of size 128K represents up to 4GB.

AMD's I/O Virtualization Technology [1] defines an IOMMU which will translate and protect memory from any DMA transfers by peripheral devices. Devices are assigned into a protection domain with a set of I/O page tables defining the allowed memory addresses. Before a DMA transfer begins, the IOMMU intercepts the access and checks its cache (IOTLB) and (if necessary) the I/O page tables for that device. The translation and isolation functions of the IOMMU may be used independently of hardware or software virtualization; however, these facilities are a natural extension to virtualization.

The AMD IOMMU is configured as a capability of a bridge or device which may be HyperTransport or PCI based. A device downstream of the AMD IOMMU in the machine topology may optionally maintain a cache (IOTLB) of its own address translations. An IOMMU may also be incorporated into a bridge downstream of another IOMMU capable bridge. Both topologies form scalable networks of distributed translations. The page structures used by the IOMMU are maintained in system memory by hypervisors or privileged OS's.

The AMD IOMMU can be used in conjunction with or in place of of the GART or DEV. While GART is limited to a 2GB translation window, the AMD IOMMU can translate accesses to all physical memory.

Intel Virtualization Technology for Directed I/O (VT-d)

Intel Virtualization Technology for Directed I/O Architecture provides DMA remapping hardware that adds support for isolation of device accesses to memory as well as translation functionality [2]. The DMA remapping hardware intercepts device attempts to access system memory. Then it uses I/O page tables to determine whether the access is allowed and its intended location. The translation structure is unique to an I/O device function (PCI bus, device, and function) and is based on a multilevel page table. Each I/O device is given the DMA virtual address space same as the physical address space, or a purely virtual address space defined by software. The DMA remapping hardware uses a context-entry table that is indexed by PCI bus, device and function to find the root of the address translation table. The hardware may cache context-entries as well as the effective translations (IOTLB) to minimize the overhead incurred for fetching them from memory. DMA remapping faults detected by the hardware are processed by logging the fault information and reporting the faults to software through a fault event (interrupt).

IBM IOMMUs: Calgary, DART, and Cell

IBM's Calgary PCI-X bridge chips provide hardware IOMMU functionality to both translate and isolate. Translations are defined by a set of Translation Control Entries (TCEs) in a table in system memory. The table can be considered an array where the index is the page number in the bus address space and the TCE at that index describes the physical page number in system memory. The TCE may also contain additional information such as DMA direction access rights and specific devices (or device groupings) that each translation can be considered valid. Calgary provides a unique bus address space to all devices behind each PCI Host Bridge (PHB). The TCE table can be large enough to cover 4GB of device accessible memory. Calgary will fetch translations as appropriate and cache them locally in a manner similar to a TLB, or IOTLB. The IOTLB, much like the TLB on an MMU, provides a software accessible mechanism that can invalidate cache entries as the entries in system memory are modified. Addresses above the 4GB boundary are accessible using PCI DAC commands. If these commands originate from the device and are permitted, they will bypass the TCE translation. Calgary ships in some IBM System P and System X systems.

IBM's CPC925 (U3) northbridge, which can be found on JS20/21 Blades and Apple G5 machines, provides IOMMU mechanisms using a DMA Address Relocation Table (DART). DART is similar to the Calgary TCE table, but differs in that the entries only track validity rather than access rights. As with the Calgary, the U3 maintains an IOTLB and provides a software-accessible mechanism for invalidating entries.

The Cell Processor has a translation- and isolation-capable IOMMU implemented on chip. Its bus address space uses a segmented translation model that is compatible with the MMU in the PowerPC core (PPE). This two-level approach not only allows for efficient user level device drivers, but also allows applications running on the Synergistic Processing Engine (SPE) to interact with devices directly. The Cell IOMMU maintains two local caches—one for caching segment entries and another for caching page table entries, the IOSLB and IOTLB, respectively. Each has a separate software accessible mechanism to invalidate entries. However, all entries in both caches are software accessible, so it is possible to program all translations directly in the caches, increasing the determinism of the translation stage.

2 Linux IOMMU support and the DMA mapping API

Linux runs on many different platforms. Those platforms may have a hardware IOMMU emulation such as SWIOTLB, or direct hardware access. The software that enables these IOMMUs must abstract its internal, device-specific DMA mapping functions behind the generic DMA API. In order to write generic, platform-independent drivers, Linux abstracts the IOMMU details inside a common API, known as the "DMA" or "DMA mapping" API [7] [8]. As long as the implementation conforms to the semantics of the DMA API, a well written driver that is using the DMA API properly should "just work" with any IOMMU.

Prior to this work, Linux's x86-64 architecture included three DMA API implementations: NOMMU, SWIOTLB, and GART. NOMMU is a simple, architecture-independent implementation of the DMA API. It is used when the system has neither a hardware IOMMU nor software emulation. All it does is return the physical memory address for the memory region it is handed as the DMA address for the adapter to use.

Linux includes a software implementation of an IOMMU's translation function, called SWIOTLB. SWIOTLB was first introduced in `arch/ia64` [3] and is used today by both IA64 and x86-64. It provides translation through a technique called *bounce buffering*. At boot time, SWIOTLB sets aside a large physically contiguous memory region (the *aperture*), which is off-limits to the OS. The size of the aperture is configurable and ranges from several to hundreds of megabytes. SWIOTLB uses this aperture as a location for DMAs that need to be remapped to system memory higher than the 4GB boundary. When a driver wishes to DMA to a memory region, the SWIOTLB code checks the system memory address of that region. If it is directly addressable by the adapter, the DMA address of the region is returned to the driver and the adapter DMAs there directly. If it is not, SWIOTLB allocates a "bounce buffer" inside the aperture, and returns the bounce buffer's DMA address to the driver. If the requested DMA operation is a DMA read

(read from memory), the data is copied from the original buffer to the bounce buffer, and the adapter reads it from the bounce buffer's memory location. If the requested DMA operation is a write, the data is written by the adapter to the bounce buffer, and then copied to the original buffer.

SWIOTLB treats the aperture as an array, and during a DMA allocation it traverses the array searching for enough contiguous empty slots in the array to satisfy the request using a next-fit allocation strategy. If it finds enough space to satisfy the request, it passes the location within the aperture for the driver to perform DMA operations. On a DMA write, SWIOTLB performs the copy ("bounce") once the driver unmaps the IO address. On a DMA read or bidirectional DMA, the copy occurs during the mapping of the memory region. Synchronization of the bounce buffer and the memory region can be forced at any time through the various `dma_sync_xxx` function calls.

SWIOTLB is wasteful in CPU operations and memory, but is the only way some adapters can access all memory on systems without an IOMMU. Linux always uses SWIOTLB on IA64 machines, which have no hardware IOMMU. On x86-64, Linux will only use SWIOTLB when the machine has greater than 4GB memory and no hardware IOMMU (or when forced through the `iommu=force` boot command line argument).

The only IOMMU that is specific to x86-64 hardware is AMD's GART. GART's implementation works in the following way: the BIOS (or kernel) sets aside a chunk of contiguous low memory (the *aperture*), which is off-limits to the OS. There is a single aperture for all of the devices in the system, although not every device needs to make use of it. GART uses addresses in this aperture as the IO addresses of DMAs that need to be remapped to system memory higher than the 4GB boundary. The GART Linux code keeps a list of the used buffers in the aperture via a bitmap. When a driver wishes to DMA to a buffer, the code verifies that the system memory address of the buffer's memory falls within the device's DMA mask. If it does not, then the GART code will search the aperture bitmap for an opening large enough to satisfy the number of pages spanned by the DMA mapping request. If it finds the required number of contiguous pages, it programs the appropriate remapping (from the aperture to the original buffer) in the IOMMU and returns the DMA address within the aperture to the driver.

We briefly described seven different IOMMU designs. AMD's GART and IBM's DART provide translation but not isolation. Conversely, AMD's Device Exclusion Vector provides isolation but not translation. IBM's Calgary and Cell are the only two architectures available today which provide both translation and isolation. However, AMD's and Intel's forthcoming IOMMUs will soon be providing these capabilities on most x86 machines.

3 Xen IOMMU support

Xen [5] [6] is a virtual machine monitor for x86, x86-64, IA64, and PowerPC that supports execution of multiple guest operating systems on the same physical machine with high performance and resource isolation. Operating systems running under Xen are either paravirtualized (their source code is modified in order to run under a hypervisor) or fully virtualized (source code was designed and written to run on bare metal and has not been modified to run under a hypervisor). Xen makes a distinction between "physical" (interchangeably referred to as "pseudo-physical") frames and machine frames. An operating system running under Xen runs in a contiguous "physical" address space, spanning from physical address zero to end of guest "physical" memory. Each guest "physical" frame is mapped to a host "machine" frame. Naturally, the physical frame number and the machine frame number will be different most of the time.

Xen has different uses for IOMMU than traditional Linux. Xen virtual machines may straddle or completely reside in system memory over the 4GB boundary. Additionally, Xen virtual machines run with a physical address space that is not identity mapped to the machine address space. Therefore, Xen would like to utilize the IOMMU so that a virtual machine with direct device access need not be aware of the physical to machine translation, by presenting an IO address space that is equivalent to the physical address space. Additionally, Xen would like virtual machines with hardware access to be isolated from other virtual machines.

In theory, any IOMMU driver used by Linux on bare metal could also be used by Linux under Xen after being suitably adapted. The changes required depend on the specific IOMMU, but in general the modified IOMMU driver would need to map from PFNs to MFNs and allocate a machine contiguous aperture rather than a pseudo-physically contiguous aperture. In practice, as of Xen's 3.0.0 release, only a modified version of SWIOTLB is supported.

Xen's controlling domain (dom0) always uses a modified version of SWIOTLB. Xen's SWIOTLB serves two purposes. First, since Xen domains may reside in system memory completely above the 4GB mark, SWIOTLB provides a machine-contiguous aperture below 4GB. Second, since a domain's pseudo-physical memory may not be machine contiguous, the aperture provides a large machine contiguous area for bounce buffers. When a stock Linux driver running under Xen makes a DMA API call, the call always goes through dom0's SWIOTLB, which makes sure that the returned DMA address is below 4GB if necessary and is machine contiguous. Naturally, going through SWIOTLB on every DMA API call is wasteful in CPU cycles and memory and has a non-negligible performance cost. GART or Calgary (or any other suitably capable hardware IOMMU) could be used to do in hardware what SWIOTLB does in software, once the necessary support is put in place.

One of the main selling points of virtualization is machine consolidation. However, some systems would like to access hardware directly in order to achieve maximal performance. For example, one might want to put a database virtual machine and a web server virtual machine on the same physical machine. The database needs fast disk access and the web server needs fast network access. If a device error or system security compromise occurs in one of the virtual machines, the other is immediately vulnerable. Because of this need for security, there is a need for software or hardware device isolation.

Xen supports the ability to allocate different physical devices to different virtual machines (multiple "driver domains" [10]). However, due to the architectural limitations of most PC hardware, notably the lack of an isolation capable IOMMU, this cannot be done securely. In effect, any domain that has direct hardware access is considered "trusted." For some scenarios, this can be tolerated. For others (e.g., a hosting service that wishes to run multiple customers virtual machines on the same physical machine), this is completely unacceptable.

Xen's grant tables are a software solution to the lack of suitable hardware for isolation. Grant tables provide a method to share and transfer pages of data between domains. They give (or "grant") other domains access to pages in the system memory allocated to the local domain. These pages can be read, written, or exchanged (with the proper permission) for the purpose of providing a fast and secure method for domains to receive indirect access to hardware.

How does data get from the hardware to the local domain that wishes to make use it, when only the driver domain can access the hardware directly? One alternative would be for the driver domain to always DMA into its own memory, and then pass the data to the local domain. Grant tables provide a more efficient alternative by letting driver domains DMA directly into pages in the local domain's memory. However, it is only possible to DMA into pages specified within the grant table. Of course, this is only significant for non-privileged domains (as privileged domains could always access the memory of non-privileged domains). Grant tables have two methods for allowing access to remote pages in system memory: shared pages and page flipping.

For shared pages, a driver in the local domain's kernel will advertise a page to be shared via a hypervisor function call ("hypercall" or "hcall"). The hcall notifies the hypervisor that other domains are allowed to access this page. The local domain then passes a grant table reference ID to the remote domain it is "granting" access to. Once the remote domain is finished, the local domain removes the grant. Shared pages are used by block devices and any other device that receives data synchronously.

Network devices, as well as any other device that receives data asynchronously, use a method known as *page flipping*. When page flipping, a driver in the local domain's kernel will advertise a page to be transferred. This call notifies the hypervisor that other domains can receive this page. The local domain then transfers the page to the remote domain and takes a free page (via producer/consumer ring).

Incoming network packets need to be inspected before they can be transferred, so that the intended destination can be deduced. Since block devices already know which domain requested data to be read, there is no need to inspect the data prior to sending it to its intended domain. Newer networking technologies (such as RDMA NICs and Infiniband) know when a packet is received from the wire for which domain is it destined and will be able to DMA it there directly.

Grant tables, like SWIOTLB, are a software implementation of certain IOMMU functionality. Much like how SWIOTLB provides the translation functionality of an IOMMU, grant tables provide the isolation and protection functionality. Together they provide (in software) a fully functional IOMMU (i.e., one that provides both translation and isolation). Hardware acceleration of grant tables and SWIOTLB is possible, provided a suitable hardware IOMMU exists on the platform, and is likely to be implemented in the future.

4 Virtualization: IOMMU design requirements and open issues

Adding IOMMU support for virtualization raises interesting design requirements and issues. Regardless of the actual functionality of an IOMMU, there are a few basic design requirements that it must support to be useful in a virtualized environment. Those basic design requirements are: memory isolation, fault isolation, and virtualized operating system support.

To achieve memory isolation, an operating system or hypervisor should not allow one virtual machine with direct hardware access to cause a device to DMA into an area of physical memory that the virtual machine does not own. Without this capability, it would be possible for any virtual machine to have access to the memory of another virtual machine, thus precluding running an untrusted OS on any virtual machine and thwarting basic virtualization security requirements.

To achieve fault isolation, an operating system or hypervisor should not allow a virtual machine that causes a bad DMA (which leads to a translation error in the IOMMU) to affect other virtual machines. It is acceptable to kill the errant virtual machine or take its devices off-line, but it is not acceptable to kill other virtual machines (or the entire physical machine) or take devices that the errant virtual machines does not own offline.

To achieve virtualized operating system support, an operating system or hypervisor needs to support para-virtualized operating systems, fully virtualized operating systems that are not IOMMU-aware, and fully virtualized IOMMU-aware operating systems. For para-virtualized OS's, the IOMMU support should mesh in seamlessly and take advantage of the existing OS IOMMU support (e.g., Linux's DMA API). For fully virtualized but not IOMMU-aware OS's, it should be possible for control tools to construct IOMMU translation tables that mirror the OS's pseudo-physical to machine mappings. For fully virtualized IOMMU aware operating systems, it should be possible to trap, validate, and establish IOMMU mappings such that the semantics the operating system expects with regards to the IOMMU are maintained.

There are several outstanding issues and open questions that need to be answered for IOMMU support. The first and most critical question is: "who owns the IOMMU?" Satisfying the isolation requirement requires that the IOMMU be owned by a trusted entity that will validate every map and unmap operation. In Xen, the only trusted entities are the hypervisor and privileged domains (i.e., the hypervisor and dom0 in standard configurations), so the IOMMU must be owned by either the hypervisor or a trusted domain. Mapping and unmapping entries into the IOMMU is a frequent, fast-path operation. In order to impose as little overhead as possible, it will need to be done in the hypervisor. At the same time, there are compelling reasons to move all hardware-related operations outside of the hypervisor. The main reason is to keep the hypervisor itself small and ignorant of any hardware details except those absolutely essential, to keep it maintainable and verifiable. Since dom0 already has all of the required IOMMU code for running on bare metal, there is little point in duplicating that code in the hypervisor.

Even if mapping and unmapping of IOMMU entries is done in the hypervisor, should dom0 or the hypervisor initialize the IOMMU and perform other control operations? There are arguments both ways. The argument in favor of the hypervisor is that the hypervisor already does some IOMMU operations, and it might as well do the rest of them, especially if no clear-cut separation is possible. The arguments in favor of dom0 are that it can utilize all of the bare metal code that it already contains.

Let us examine the simple case where a physical machine has two devices and two domains with direct hardware access. Each device will be dedicated to a separate domain. From the point of view of the IOMMU, each device has a different IO address space, referred to simply as an "IO space." An IO space is a virtual address space that has a distinct translation table. When dedicating a device to a domain, we either establish the IO space *a priori* or let the domain establish mappings in the IO space that will point to its machine pages as it needs them. IO spaces are created when a device is granted to a domain, and are destroyed when the device is brought offline (or when the domain is destroyed). A trusted entity grants access to devices, and therefore necessarily creates and grants access to their IO spaces. The same trusted entity can revoke access to devices, and therefore revoke access and destroy their IO spaces.

There are multiple considerations that need to be taken into account when designing an IOMMU interface. First, we should differentiate between the administrative interfaces that will be used by control and management tools, and "data path" interfaces which will be used by unprivileged domains. Creating and destroying an IO space is an administrative interface; mapping a machine page is a data path operation.

Different hardware IOMMUs have different characteristics, such as different degrees of device isolation. They might support no isolation (single global IO address space for all devices in the system), isolation between different busses (IO address space per PCI bus), or isolation on the PCI Bus/Device/Function (BDF) level (i.e., a separate IO address space for each logical PCI device function). The IO space creation interface should expose the level of isolation that the underlying hardware is capable of, and should support any of the above isolation schemes. Exposing a finer-grained isolation than the hardware is capable of could lead software to a false sense of security, and exposing a coarser grained isolation would not be able to fully utilize the capabilities of the hardware.

Another related question is whether several devices should be able to share the same IO address space, even if the hardware is capable of isolating between them. Let us consider a fully virtualized operating system that is not IOMMU aware and has several devices dedicated to it. Since the OS is not capable of utilizing isolation between these devices and each IO space consumes a small, yet non-negligible amount of memory for its translation tables, there is no point in giving each device a separate IO address space. For cases like this, it would be beneficial to share the same IO address space among all devices dedicated to a given operating system.

We have established that it may be beneficial for multiple devices to share the same IO address space. Is it likewise beneficial for multiple consumers (domains) to share the same IO address space? To answer this question, let us consider a smart IO adapter such as an Infiniband NIC. An IB NIC handles its own translation needs and supports many more concurrent consumers than PCI allows. PCI dedicates 3 bits for different "functions" on the same device (8 functions in total) whereas IB supports 24 bits of different consumers (millions of consumers). To support such "virtualization friendly" adapters, one could run with translation disabled in the IOMMU, or create a single IO space and let multiple consumers (domains) access it.

Since some hardware is only capable of having a shared IO space between multiple non-cooperating devices, it is beneficial to be able to create several logical IO spaces, each of which is a window into a single large "physical IO space." Each device gets its own window into the shared address space. This model only provides "statistical isolation." A driver programming a device may guess another device's window and where it has entries mapped, and if it guesses correctly, it could DMA there. However, the probability of its guessing correctly can be made fairly small. This mode of operation is not recommended, but if it's the only mode the hardware supports…

Compared to creation of an IO space, mapping and unmapping entries in it is straightforward. Establishing a mapping requires the following parameters:

- A consumer needs to specify which IO space it wants to establish a mapping in. Alternatives for for identifying IO spaces are either an opaque, per-domain "IO space handle" or the BDF that this IO space translates for.

- The IO address in the IO address space to establish a mapping at. The main advantage of letting the domain pick the IO address it that it has control over how IOMMU mappings are allocated, enabling it to optimize their allocation based on its specific usage scenarios. However, in the case of shared IO spaces, the IO address the device requests may not be available or may need to be modified. A reasonable compromise is to make the IO address a "hint" which the hypervisor is free to accept or reject.
- The access permissions for the given mapping in the IO address space. At a minimum, any of `none`, `read only`, `write only`, or `read write` should be supported.
- The size of the mapping. It may be specified in bytes for convenience and to easily support different page sizes in the IOMMU, but ultimately the exact size of the mapping will depend on the specific page sizes the IOMMU supports.

To reduce the number of required hypercalls, the interface should support multiple mappings in a single hypervisor call (i.e., a "scatter gather list" of mappings).

Tearing down a mapping requires the following parameters:

- The IO space this mapping is in.
- The mapping, as specified by an IO address in the IO space.
- The size of the mapping.

Naturally, the hypervisor needs to validate that the passed parameters are correct. For example, it needs to validate that the mapping actually belongs to the domain requesting to unmap it, if the IO space is shared.

Last but not least, there are a number of miscellaneous issues that should be taken into account when designing and implementing IOMMU support. Since our implementation is targeting the open source Xen hypervisor, some considerations may be specific to a Xen or Linux implementation.

First and foremost, Linux and Xen already include a number of mechanisms that either emulate or complement hardware IOMMU functionality. These include SWIOTLB, grant tables, and the PCI frontend / backend drivers. Any IOMMU implementation should "play nicely" and integrate with these existing mechanisms, both on the design level (i.e., provide hardware acceleration for grant tables) and on the implementation level (i.e., do not duplicate common code).

One specific issue that must be addressed stems from Xen's use of page flipping. Pages that have been mapped into the IOMMU must be pinned as long as they are resident in the IOMMU's table. Additionally, any pages that are involved in IO may not be relinquished by a domain (e.g., by use of the balloon driver).

Devices and domains may be added or removed at arbitrary points in time. The IOMMU support should handle "garbage collection" of IO spaces and pages mapped in IO when the domain or domains that own them die or the device they map is removed. Likewise, hotplugging of new devices should also be handled.

5 Calgary IOMMU Design and Implementation

We have designed and implemented IOMMU support for the Calgary IOMMU found in high-end IBM System X servers. We developed it first on bare metal Linux, and then used the bare metal implementation as a stepping-stone to a "virtualization enabled" proof-of-concept implementation in Xen. This section describes both implementations. It should be noted that Calgary is an isolation-capable IOMMU, and thus provides isolation between devices residing on different PCI Host Bridges. This capability is directly beneficial in Linux even without a hypervisor for its RAS capabilities. For example, it could be used to isolate a device in its own IO space while developing a driver for it, thus preventing DMA related errors from randomly corrupting memory or taking down the machine.

5.1 x86-64 Linux Calgary support

The Linux implementation is included at the time of this writing in 2.6.16-mm1. It is composed of several parts: initialization and detection code, IOMMU specific code to map and unmap entries, and a DMA API implementation.

The bring-up code is done in two stages: detection and initialization. This is due to the way the x86-64 arch-specific code detects and initializes IOMMUs. In the first stage, we detect whether the machine has the Calgary chipset. If it does, we mark that we found a Calgary IOMMU, and allocate large contiguous areas of memory for each PCI Host Bridge's translation table. Each translation table consists of a number of entries that total the addressable range given to the device (in page size increments). This stage uses the bootmem allocator and happens before the PCI subsystem is initialized. In the second stage, we map Calgary's internal control registers and enable translation on each PHB.

The IOMMU requires hardware-specific code to map and unmap DMA entries. This part of the code implements a simple TCE allocator to "carve up" each translation table to different callers, and includes code to create TCEs (Translation Control Entries) in the format that the IOMMU understands and writes them into the translation table.

Linux has a DMA API interface to abstract the details of exactly how a driver gets a DMA'able address. We implemented the DMA API for Calgary, which allows generic DMA mapping calls to be translated to Calgary specific DMA calls. This existing infrastructure enabled the Calgary Linux code to be more easily hooked into Linux without many non-Calgary specific changes.

The Calgary code keeps a list of the used pages in the translation table via a bitmap. When a driver make a DMA API call to allocate a DMA address, the code searches the bitmap for an opening large enough to satisfy the DMA allocation request. If it finds enough space to satisfy the request, it updates the TCEs in the translation table in main memory to let the DMA through. The offset of those TCEs within the translation table is then returned to the device driver as the DMA address to use.

5.2 Xen Calgary support

Prior to this work, Xen did not have any support for isolation-capable IOMMUs. As explained in previous sections, Xen does have software mechanisms (such as SWIOTLB and grant tables) that emulate IOMMU-related functionality, but does not have any hardware IOMMU support, and specifically does not have any isolation-capable hardware IOMMU support.

We added proof-of-concept IOMMU support to Xen. The IOMMU support is composed of a thin "general IOMMU" layer, and hardware IOMMU specific implementations. At the moment, the only implementation is for the Calgary chipset, based on the bare-metal Linux Calgary support. As upcoming IOMMUs become available, we expect more hardware IOMMU implementations to show up.

It should be noted that the current implementation is proof-of-concept and is subject to change as IOMMU support evolves. In theory it targets numerous IOMMUs, each with distinct capabilities, but in practice it has only been implemented for the single isolation-capable IOMMU that is currently available. We anticipate that by the time you read this, the interface will have changed to better accommodate other IOMMUs.

The IOMMU layer receives the IOMMU related hypercalls (both the "management" hcalls from dom0 and the IOMMU map/unmap hcalls from other domains) and forwards them to the IOMMU specific layer. The following hcalls are defined:

- `iommu_create_io_space` – this call is used by the management domain (dom0) to create a new IO space that is attached to specific PCI BDF values. If the IOMMU supports only bus level isolation, the device and function values are ignored.
- `iommu_destroy_io_space` – this call is used to destroy an IO space, as identified by a BDF value.

Once an IO space exists, a domain can ask to map and unmap translation entries in its IOMMU using the following calls:

- `u64 do_iommu_map(u64 ioaddr, u64 mfn, u32 access, u32 bdf, u32 size);`
- `int do_iommu_unmap(u64 ioaddr, u32 bdf, u32 size);`

When mapping an entry, the domain passes the following parameters:

- `ioaddr` – The address in the IO space that the domain would like to establish a mapping at. This is a hint; the hypervisor is free to use it or ignore it and return a different IO address.
- `mfn` – The machine frame number that this entry should map. In the current Xen code base, a domain running on x86-64 and doing DMA is aware of the physical/machine translation, and thus there is no problem with passing the MFN. In future implementations this API will probably change to pass the domain's PFN instead.
- `access` – This specifies the Read/Write permission of the entry (*read* here refers to what the device can do—whether it can only read from memory, or can write to it as well).
- `bdf` – The PCI Bus/Device/Function of the IO space that we want to map this in. This parameter might be changed in later revisions to an opaque IO-space identifier.
- `size` – How large is this entry? The current implementation only supports a single IOMMU page size of 4KB, but we anticipate that future IOMMUs will support large page sizes.

The return value of this function is the IO address where the entry has been mapped.

When unmapping an entry, the domain passes the BDF, the IO address that was returned and the size of the entry to be unmapped. The hypervisor validates the parameters, and if they validate correctly, unmaps the entry.

An isolation-capable IOMMU is likely to either have a separate translation table for different devices, or have a single, shared translation table where each entry in the table is valid for specific BDF values. Our scheme supports both usage models. The generic IOMMU layer finds the right translation table to use based on the BDF, and then calls the hardware IOMMU-specific layer to map or unmap an entry in it. In the case of one domain owning an IO space, the domain can use its own allocator and the hypervisor will always use the IO addresses the domain wishes to use. In the case of a shared IO space, the hypervisor will be the one controlling IO address allocation. In this case IO address allocation could be done in cooperation with the domains, for example by adding a per domain offset to the IO addresses the domains ask for—in effect giving each domain its own window into the IO space.

6 Roadmap and Future Work

Our current implementation utilizes the IOMMU to run dom0 with isolation enabled. Since dom0 is privileged and may access all of memory anyway, this is useful mainly as a proof of concept for running a domain with IOMMU isolation enabled. Our next immediate step is to run a different, non privileged and non trusted "direct hardware access domain" with direct access to a device and with isolation enabled in the IOMMU.

Once we've done that, we plan to continue in several directions simultaneously. We intend to integrate the Calgary IOMMU support with the existing software mechanisms such as SWIOTLB and grant tables, both on the interface level and the implementation (e.g., sharing for code related to pinning of pages involved in ongoing DMAs). For configuration, we are looking to integrate with the PCI frontend and backend drivers, and their configuration mechanisms.

We are planning to add support for more IOMMUs as hardware becomes available. In particular, we look forward to supporting Intel and AMD's upcoming isolation-capable IOMMUs.

Longer term, we see many exciting possibilities. For example, we would like to investigate support for other types of translation schemes used by some devices (e.g. those used by Infiniband adapters).

We have started looking at tuning the IOMMU for different performance/reliability/security scenarios, but do not have any results yet. Most current-day machines and operating systems run without any isolation, which in theory should give the best performance (least overhead on the DMA path). However, IOMMUs make it possible to perform scatter-gather coalescing and bounce buffer avoidance, which could lead to increased overall throughput.

When enabling isolation in the IOMMU, one could enable it selectively for "untrusted" devices, or for all devices in the system. There are many trade-offs that can be made when enabling isolation: one example is static versus dynamic mappings, that is, mapping the entire OS's memory into the IOMMU up front when it is created (no need to make map and unmap hypercalls) versus only mapping those pages that are involved in DMA activity. When using dynamic mappings, what is the right mapping allocation strategy? Since every IOMMU implements a cache of IO mappings (an IOTLB), we anticipate that the IO mapping allocation strategy will have a direct impact on overall system performance.

7 Conclusion: Key Research and Development Challenges

We implemented IOMMU support on x86-64 for Linux and have proof-of-concept IOMMU support running under Xen. We have shown that it is possible to run virtualized and non-virtualized operating systems on x86-64 with IOMMU isolation. Other than the usual woes associated with bringing up a piece of hardware for the first time, there are also interesting research and development challenges for IOMMU support.

One question is simply how can we build better, more efficient IOMMUs that are easier to use in a virtualized environment? The upcoming IOMMUs from IBM, Intel, and AMD have unique capabilities that have not been explored so far. How can we best utilize them and what additional capabilities should future IOMMUs have?

Another open question is whether we can use the indirection IOMMUs provide for DMA activity to migrate devices that are being accessed directly by a domain, without going through an indirect software layer such as the backend driver. Live virtual machine migration ("live" refers to migrating a domain while it continues to run) is one of Xen's strong points [9], but at the moment it is mutually incompatible with direct device access. Can IOMMUs mitigate this limitation?

Another set of open question relate to the ongoing convergence between IOMMUs and CPU MMUs. What is the right allocation strategy for IO mappings? How to efficiently support large pages in the IOMMU? Does the fact that some IOMMUs share the CPU's page table format (e.g., AMD's upcoming IOMMU) change any fundamental assumptions?

What is the right way to support fully virtualized operating systems, both those that are IOMMU-aware, and those that are not?

We continue to develop Linux and Xen's IOMMU support and investigate these questions. Hopefully, the answers will be forthcoming by the time you read this.

8 Legal

Any statements about support or other commitments may be changed or canceled at any time without notice. All statements regarding future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. Information is provided "AS IS" without warranty of any kind. The information could include technical inaccuracies or typographical errors. Improvements and/or changes in the product(s) and/or the program(s) described in this publication may be made at any time without notice.

References

[1] *AMD I/O Virtualization Technology (IOMMU)* Specification, 2006, `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/34434.pdf`.

[2] *Intel Virtualization Technology for Directed I/O Architecture Specification*, 2006, `ftp://download.intel.com/technology/computing/vptech/Intel(r)_VT_for_Direct_IO.pdf`.

[3] *IA-64 Linux Kernel: Design and Implementation*, by David Mosberger and Stephane Eranian, Prentice Hall PTR, 2002, ISBN 0130610143.

[4] *Software Optimization Guide for the AMD64 Processors*, 2005, `http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25112.PDF`.

[5] *Xen and the Art of Virtualization*, by B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer, in Proceedings of the 19th ASM Symposium on Operating Systems Principles (SOSP), 2003.

[6] *Xen 3.0 and the Art of Virtualization*, by I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick, in Proceedings of the 2005 Ottawa Linux Symposium (OLS), 2005.

[7] *Documentation/DMA-API.txt*.

[8] *Documentation/DMA-mapping.txt*.

[9] *Live Migration of Virtual Machines*, by C. Clark, K. Fraser, S. Hand, J. G. Hanseny, E. July, C. Limpach, I. Pratt, A. Warfield, in Proceedings of the 2nd Symposium on Networked Systems Design and Implementation, 2005.

[10] *Safe Hardware Access with the Xen Virtual Machine Monitor*, by K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, M. Williamson, in Proceedings of the OASIS ASPLOS 2004 workshop, 2004.

[11] *PCI Special Interest Group* `http://www.pcisig.com/home`.

Towards a Highly Adaptable Filesystem Framework for Linux

Suparna Bhattacharya
Linux Technology Center
IBM Software Lab, India
suparna@in.ibm.com

Dilma Da Silva
IBM T.J. Watson Research Center, USA
dilmasilva@us.ibm.com

Abstract

Linux® is growing richer in independent general purpose file systems with their own unique advantages, however, fragmentation and divergence can be confusing for users. Individual file systems are also adding an expanding number of options (e.g. ext3) and variations (e.g. reiser4 plugins) to satisfy new requirements. Both of these trends indicate a need for improved flexibility in file system design to benefit from the best of all worlds. We explore ways to address this need, using as our basis, KFS (K42 file system), a research file system designed for fine-grained flexibility.

KFS aims to support a wide variety of file structures and policies, allowing the representation of a file or directory to change on the fly to adapt to characteristics not well known a priori, e.g. list-based to tree-based, or small to large directory layouts. It is not intended as yet another file system for Linux, but as a platform to study trade-offs associated with adaptability and evaluate new algorithms before incorporation on an established file system. We hope that ideas and lessons learnt from the experience with KFS will be beneficial for Linux file systems to evolve to be more adaptable and for the VFS to enable better building-block-based code sharing across file systems.

1 Introduction

The Linux 2.6 kernel includes over 40 filesystems, about 0.6 million lines of code in total. The large number of Linux filesystems as well as their sheer diversity is a testament to the power and flexibility of the Linux VFS. This has enabled Linux to support a wide range of existing file system formats and protocols. Interestingly, this has also resulted in a growing number of new file systems that have been developed for Linux. The 2.5 development series saw the inclusion of four(ext3, reiserFS, JFS, and XFS) general purpose journalling filesystems, while in recent times multiple cluster filesystems have been submitted to the mainline kernel, providing users a slew of alternatives to choose from. Allowing multiple independent general purpose filesystems to co-exist has also had the positive effect of enabling each to innovate in parallel within its own space making different trade-offs and evolving across multiple production releases. New file systems have a chance to prove themselves out in the real world, letting time pick the best one rather than standardize on one single default filesystem[6].

At the same time, the multiplicity of filesystems that essentially address very similar needs also sometimes leads to unwarranted fragmentation and divergence from the perspective of users who may find themselves faced with complex administrative choices with associated lock-in to a file system format chosen at a certain point in time. This is probably one reason why most users tend to simply adopt the default file system provided by their Linux distribution (i.e. ext3 or reiserfs), despite the availability of possibly more advanced filesystems like XFS and JFS, which might have been more suitable for their primary workloads. Each individual filesystem has been evolving to expand its capabilities by adding support for an increasing number of options and variations, to satisfy new requirements, and make continuous improvements while also maintaining compatibility.

We believe that these trends point to a need for a framework for a different kind of flexibility in file system design for the Linux kernel, one that allows continuous adaptability to evolving requirements, but reduces duplicate effort while providing users the most appropriate layout and capabilities for the workload and file access patterns they are running.

This work has two main goals: (1) to investigate how far a design centered on supporting dynamic customization of services can help to address Linux's needs for flexible file systems and (2) to explore performance benefits and trade-offs involved in a design for flexibility.

The basis of our exploration is the HFS/KFS research effort started a decade ago. In the Hurricane File System (HFS), the support for dynamic alternatives for file layout were motivated by the scalability requirements of the workloads targetted by the Hurricane operating system project. The K42 File System (KFS) built on the flexibility basis of HFS, expanding it by encorporating the architectural principles in the K42 Research Operating System project. While KFS was designed as a separate filesystem of its own, in this work we explore what it would take to apply similar techniques to existing filesystems, where such fine-grained flexibility was not an original design consideration. We also update KFS to work with Linux 2.6, aiming at carrying out experimental work that can provide concrete information about the impact of KFS's approach on addressing workload-specific performance requirements.

The rest of the paper is organized as follows: Section 2 illustrates how adaptability is currently held in Linux filesystems; Section 3 presents the basic ideas from KFS's design; Section 4 discusses KFS's potential as an adaptable filesystem framework for Linux and Section 5 describes required future work to enable this vision. Section 6 concludes.

2 Adaptability in Linux filesystems

2.1 Flexibility provided by the VFS layer

The Linux Virtual File System (VFS) [24] is quite powerful in terms of the flexibility it provides for implementing filesystems, whether disk-based filesystems, network filesystems or special purpose pseudo filesystems. This flexibility is achieved by abstracting key filesystem objects, i.e. the super block, inode and file (for both files and directories) including the address space mapping, the directory entry cache, and methods associated with each of these objects. It is possible to allow different inodes in the same filesystem to have different operation vectors. This is used, for example, by ext3 to support different journalling modes elegantly, by some types of stackable/filter filesystems to provide additional functionality to an existing filesystem, and even for specialized access modes determined by open mode, e.g. execute-in-place support. Additionally, the inclusion of extended attributes support in the VFS methods allows customizable persistent per-inode state to be maintained, which can be used to specify variations in functional behavior at an individual file level.

The second aspect of flexibility ensues from common code provided for implementation of various file operations, i.e. generic routines which can be invoked by filesystems or wrapped with additional filesystem-specific code. The bulk of interfacing between the VFS and the Virtual Memory Manager (VMM), including read-ahead logic, page-caching and access to disk blocks for block-device-based filesystems, happens in this manner. Some of these helper routines (e.g. *mpage_writepages())* accept function pointers as arguments, e.g. for specifying a filesystem specific *getblock()* routine, which also allows for potential variation of block mapping and consistency and allocation policies even within the same filesystem.

Another category of helper routines called *libfs* [4] is intended for simplifying the task of writing new virtual filesystems, though currently targeted mainly at stand-alone virtual filesystems developed for special-purpose interfacing between kernel and user space, as an alternative to using *ioctls* or new system calls.

2.2 Flexibility within individual filesystems

Over time, the need to satisfy new requirements while maintaining on-disk compatibility to the extent possible has led individual filesystems to incorporate a certain degree of variability within each filesystem specific implementation, using mount options, fcntls/ioctls, file attributes and internal layering. In this sub-section we cover a few such examples. While we limit this discussion to a few disk-based general purpose filesystems, similar approaches apply to other filesystem types as well.

We do not presently consider file systems that are not intended to be part of the mainline Linux kernel tree.

2.2.1 Ext3 options and backward compatibility

One of the often cited strengths of the ext3 filesystem is its high emphasis on backwards and forwards compatibility and dependability, even as it continues evolving to incorporate new features and enhancements. This has been achieved through a carefully designed compatibility bitmap scheme [22], the use of mount options and *tune2fs* to turn on individual features per filesystem, conversion utilities to ease migration to new incompatible features (e.g. using resize2fs-type capability), and per-file flags and attributes that can be controlled through the *chattr* command.

Three compatibility bitmaps in the super block determine if and how an old kernel would mount a filesystem with an unknown feature bit marked in each of these bitmaps: read-write (COMPAT), read-only (RO_COMPAT), and incompatible (INCOMPAT)). For safety reasons, though, the filesystem checker *e2fsck* takes the stringent approach of not touching a filesystem with an unknown feature bit even if it is in the COMPAT set, recommending the usage of a newer version of the utility instead. Backward compatibility with an older kernel is useful during a safe revert of an installation/upgrade or in enabling the disk to be mounted from other Linux systems for emergency recovery purposes. For these reasons interesting techniques have been used in the development of features like directory indexing [16] making interior index nodes look like deleted directory entries and clearing directory-indexing flags when updating directories in older kernels to ensure compatibility as far as possible. Similar considerations are being debated during the design of file pre-allocation support to avoid exposing uninitialized pre-allocated blocks if mounted by an older kernel. With the inclusion of per-inode compatibility flags, the granularity of backward compatibility support can be narrowed down to a per-file level. This may be useful during integration of extent maps.

The use of mount options and *tune2fs* makes it possible for new, relatively less established or incompatible features to be turned on optionally with explicit administrator knowledge for a few production releases before being made the default. Also, in the future, advanced feature sets may be bundled into a higher level group that signifies a generational advance of the filesystem [23]. Mount options are also used for setting consistency policies (i.e. journalling mode) on a per filesystem basis. Additionally, ext3 makes use of persistent file attributes (through the introduction of the *chattr* command), in combination with the ability to use different operation vectors for different inodes, to allow certain features/policies (e.g. full data journalling support for certain log files, preserving reservation beyond file close for slow-growing files) to be specified for individual files.

2.2.2 JFS and XFS

Although JFS [20] does not implement compatibility bitmaps as ext3 does, its on-disk layout is scalable, and backward compatibility has not been much of an issue. The on-disk directory structure was changed shortly after JFS was ported from OS/2® to Linux, causing a version bump in the super block. Since then, there has been no need to change the on-disk layout of JFS. The kernel will still support the older, OS/2-compatible, format.

JFS uses extent-based data structures and uses 40 bits to store block offsets and addresses. The on-disk layout supports various block sizes, although the kernel currently only supports a 4 KB block size. Without increasing the block size, JFS can support files and partitions up to 4 petabytes in length. B+ trees are used to implement both the extent maps and directories. Inodes are dynamically allocated as needed, and extents of free inodes are reclaimed. JFS can store file names in 16-bit unicode, translating from the code page specified by the iocharset mount option. The default is to do no translation.

JFS supports most mount options and chattr flags that ext3 does. Journaling can be temporarily disabled with the nointegrity mount flag. This is primarily intended to speed up the population of a partition, for instance, from backup media, where recovery would involve reformatting and restarting the process, with no risk of data loss.

XFS [21] also has a scalable ondisk layout, uses extent based structures, variable block sizes, dynamic inode allocation and separate allocation groups. It supports an optional real-time allocator suitable for media streaming applications.

2.2.3 Reiser4 plugins

Much like ext3 and JFS, the reiserfs v3 filesystem included in the current Linux kernel uses mount options and persistent inode attributes (set via chattr) to provide new features and variability of policies. For example, the hash function for directories can be selected by a mount option, as can some tuning of the block allocator, while tail-merging can be disabled on both a per mount point or per inode basis. It supports the same journalling modes as ext3.

The next generation of reiserfs, the reiser4 filesystem [15] (not yet in the mainline Linux kernel), includes an internal layering architecture that is intended to allow for the development for different kinds of plugins to make extensions to the filesystem without having to upgrade/format to a new version. This approach enables the addition of new features like compression, support for alternate semantics, directory ordering, security schemes, block allocation and consistency policies, and node balancing policies for different items. The stated goal of this architecture is to enable and encourage developers to build plugins to customize the filesystem with features desired by applications. From the available material at this stage, it is not clear to us yet the extent to which plugins are intended to address fine-grain flexibility or dynamic changes of on-disk data representation beyond tail formatting and item merging in dancing trees. Also, at a first glance our (possibly mistaken) perception is that reiser4 flexibility support comes with the price of complexity: the code base is large, and the interfaces appear to be tied to reiser4 internals.

2.3 Limitations of current approaches

While there is a considerable amount of flexibility within the existing framework, both at the VFS level and within individual file systems, there are some observations and issues emerging with the evolution of multiple new filesystems and new generations of filesystems that have been in existence for a while.

- Code commonality is supported at higher levels but not for lower level data manipulation, e.g. with the inclusion of extents support for ext3 there would be over 5 B+ separate tree implementations across individual filesystems.
- While the combination of per-inode operation vectors and persistent attribute flags allows for flexibility at per inode level, and alternate allocation schemes can potentially be encapsulated in the *get_blocks()* function pointer used for a given operation, there is no general framework to support different layouts for different files in a cohesive manner, to move from an old scheme to a new one in a modular fashion, or supply different meta-data or data allocation policies for a group of files, because the inode representation is typically fixed upfront.
- There is no framework for filesystems to provide their building blocks for use by other filesystems, for example even though OCFS2 [5] chose to start with a lot of code from ext3 as its base, this involved copying source code and then editing it to add all the function it needed for clustering support and scalability. As a result, extensions like 64-bit and extents support from OCFS2 can not be applied back as extensions to ext3 as an option. Because of its long history of simplicity and dependability, ext2/3 is often the preferred choice for basing experimentation for advanced capabilities [9], so the ability to specialize behaviour starting from a common code base is likely to be useful.
- Difficulty with making changes to the on-disk format for an existing file system results in implementers getting locked into supporting a change once made, and hence requires very careful consideration and make take years into real deployment especially for incompatible features. Even compatible features on ext2/3 are incompatible with older filesystem checkers.

3 Overview of KFS

KFS builds on the research done by the Hurricane File System (HFS) [14, 13, 12]. HFS was designed for (potentially large-scale) shared-memory multiprocessors, based on the principle that, in order to maximize performance for applications with diverse requirements, a file system must support a wide variety of file structures, file system policies, and I/O interfaces. As an extreme example, HFS allows a file's structure to be optimized for concurrent random-access write-only operations by 10 threads, something no other file system can do. HFS explored its flexibility to achieve better performance and scalability. It proved that its flexibility came with little processing or I/O overhead. KFS took HFS's principles further by eliminating global data structures or policies. KFS runs as a file system for the K42 [10] and Linux operating systems.

The basic aspect of KFS's enablement of fine-grained customization is that each virtual or physical resource instance (e.g., a particular file, open file instance, block allocation map) is implemented by a different set of (C++) objects. The goal of this object-oriented design is to allow each file system element to have the logical and physical representation that better matches its size and access pattern characteristics. Each element in the system can be serviced by the object that best fits its requirements; if the requirements change, the component representing the element in KFS can be replaced accordingly. Applications can achieve better performance by using the services that match their access patterns, scalability, and synchronization requirements.

When a KFS file system is mounted, the blocks on disk corresponding to the superblock are read, and a *SuperBlock* object is instantiated to represent it. A *BlockMap* object is also instantiated to represent block allocation information. Another important object instantiated at file system creation time is the *RecordMap*, which keeps the association between file system elements, their object type, and their disk location. In many traditional Unix file systems, this association is fixed and implicit: every file or directory corresponds to an inode number; inode location and inode data representation is fixed a priori. Some file systems support dynamic block allocation for inodes and a set of alternative inode representations. In KFS, instead of trying to accommodate new possibilities for representation and dynamic policies incrementally, we take the riskier route of starting with a design intended to support change and diversity of representations. KFS explores the impact of an architecture centered on supporting the design and deployment of evolving alternative representations for file system resources. The goal is to learn how far this architecture can go in supporting flexibility, and what are the trade-offs involved in this approach.

An element (file or directory) in KFS is represented in memory by two objects: one providing a logical view of the object (called Logical Server Object, or LSO), and one encapsulating its persistent characteristics (Physical Server Object, or PSO).

Figure 1 portrays the scenario where three files are instantiated: a small file, a very large file, and a file where extended allocation is being used. These files are represented by a common logical object (LSO) and by PSO objects tailored for their specific characteristics: PSOsmall, PSOextent, PSOlarge. If the small file grows, the PSOsmall is replaced by the appropriate object (e.g., PSOlarge). The *RecordMap* object is updated in order to reflect the new object type and the (potential) new file location on disk.

KFS file systems may spawn multiple disks. Figure 2 pictures a scenario where file X is being replicated on the two available disks,

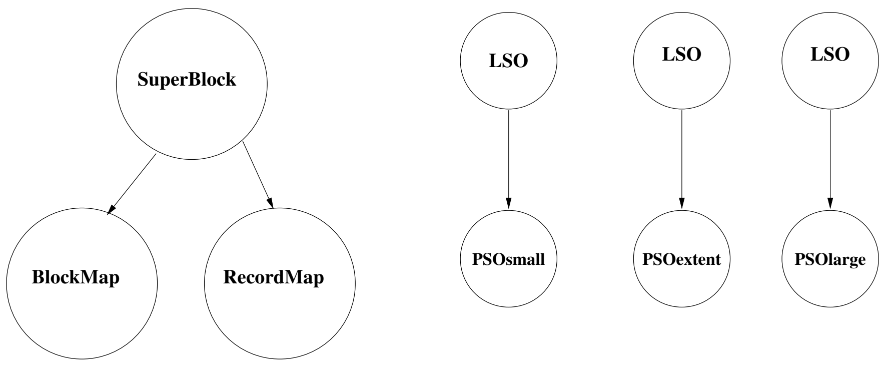


Figure 1: Objects representing files with different size and access pattern characteristics in KFS.

while file Y is being striped on the two disks in a round-robin fashion, and file Z is also being replicated, but with its content being compressed before going to disk.

In the current KFS implementation, when a file is created the choice of object implementation to be used is explicitly made by the file system code based on simplistic heuristics. Also, the user could specify intended behavior by changing values on the appropriate objects residing on /proc or use of extended attributes to provide hints about the data elements they are creating or manipulating.

As the file system evolves, compatibility with "older" formats is kept as long as the file system knows how to instantiate the object type to handle the old representation.

The performance experiments with KFS for Linux 2.4 indicate that KFS's support for flexibility doesn't result in unreasonable overheads. KFS on Linux 2.4 (with an implementation of inode and directory structures matching ext2) was found to run with a performance similar to ext2 on many workloads and 15% slower on some. These results are described in [18]. New performance data is being made available at [11] as we tune KFS's integration with Linux 2.6.

There are two ongoing efforts on using KFS's flexible design to quickly prototype and evaluate opportunities for alternative policies and data representation:

- Co-location of meta-data and data: a prototype of directory structure for embedding file attributes in directory entries where we extend the ideas proposed in [8] by doing meta-data and block allocation on a per-directory basis;

- Local vs global RecordMap structure: although KFS's design tried to avoid the use of global data structures, its initial prototype implementation has a single object (one instance of the *RecordMap* class) responsible for mapping elements onto type and disk location. As our experimentation progressed, it became clear that this data structure was hindering scalability and flexibility, and imposing performance overhead due to lack of locality between the RecordMap entry for a file and its data. Our current design explores associating different RecordMap objects with different parts of the name space.

More information about KFS can be found at [18, 7].

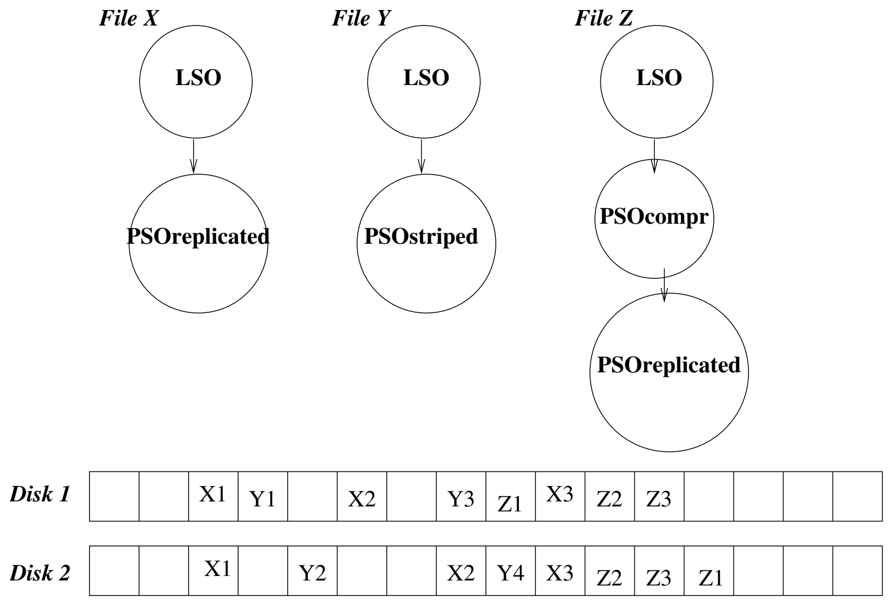


Figure 2: KFS objects and block allocation for files X (replicated; blocks X1, X2, X3), Y (striped; blocks Y1, Y2, Y3, Y4), and Z (compressed and replicated; blocks Z1, Z2, Z3).

4 Learnings from KFS towards an adaptable filesystem framework for Linux

What possibilities does the experience with KFS suggest towards addressing some of the concerns discussed in section 2.3? Intuitively, it appears that the ideas originating from HFS, and developed further in KFS, of a bottom-up building-block-based, fine-grained approach to flexibility for experimenting with specialized behaviour and data organization on a per-element basis, could elegantly complement the current VFS approach of abstracting higher levels of commonality that allows an abundance of filesystems. While KFS was developed as a separate filesystem, we do not believe that adding yet another filesystem to Linux is the right answer. Instead developing simple methods of applying KFS-like flexibility to existing filesystems incrementally, while possibly non-trivial, may be a promising approach to pursue.

4.1 Switching formats

While many of the file systems mentioned in Section 2 (ext2/3, reiserfs, JFS, XFS, OCFS2) are intended to be general-purpose file systems, each appears to have its own sweet-spot usage scenarios or access patterns that sets it apart from the rest. Various comparison charts [1, 17, 19] exist that highlight the strengths and weaknesses of these alternatives to enable administrators to make the right choice for their systems when they format their disks at the time of first installation. However, predicting the nature of workloads ahead of time, especially when mixed access patterns may apply to different files in the same filesystem at different times, is difficult. Unlike configuration options or run-time tunables, switching a choice of on-disk format on a system is highly disruptive and resoure consuming. With an adaptable framework that can alter representations on a per-element basis in response to significant changes in access patterns, living with a less than optimal solution as a result of being locked into a given on-disk filesystem representation would no longer be necessary.

We have described earlier (section 3) that in KFS it is possible to create new formats and work with them, with other formats being simultaneously active as well. This is possible due to (1) the association between a Logical Storage Object (LSO) and its associated Physical Storage Object (PSO) is not fixed, and (2) the implementation of local rather than global control structures for different types of resource allocation. We plan to experiment with abstracting this mechanism so that it can be used by existing filesystems, for example for upgrading to new file representations like 64-bit and extents support in ext3 including the new multi-block allocator implementation from Alex Tomas [2]. We would like to compare the results with the current approach in ext3 for achieving the format switch, which relies on per-inode flags and alternate inode operation vectors.

4.2 Evaluation of alternate layouts and policies

The ability to switch formats and layout policies enables KFS to provide a platform for determining optimal layout and allocation policies for a filesystem through ongoing experimentation and comparative evaluation. Typically, layout decisions are very closely linked to the particular context in which the file system is intended to be used, e.g. considering underlying media, nature of application workloads, data access patterns, available space, filesystem aging/fragmentation, etc. With the mechanisms described in the previous sub-section in place, we intend to demonstrate performance advantages over the long run from being able to choose and switch from alternative representations, for example from a very small file oriented representation with data embedded within the inode, to direct, indirect block mapping, to extents maps based on file size, distribution and access patterns.

4.3 Assessment of overheads imposed by flexibility

It is said that any problem in computer science can be solved by adding an extra level of indirection. The only problem is that indirections do not come for free, especially if it involves extra disk seeks to perform an operation. It is for this reason that ext2/3, for example, attempts to allocate indirect blocks contiguously with the data blocks they map to, and why support for extended attributes storage in large inodes has been demonstrated to deliver significant performance gains for Samba workloads [3] compared to storage of attributes in a separate block. This issue has been a primary design consideration for KFS. The original HFS/KFS effort has been specifically conceptualized with a view towards enabling high performance and scalability through fine-grained flexibility, rather than with an intent of adding layers of high level semantic features.

Initial experiments with KFS seem to indicate that the benefits of flexibility outweigh overheads, given a well-designed meta-data caching mechanism and right choice of building blocks where indirections go straight to the most appropriate on-disk objects when a file element is first looked up. The ability to have entirely different and variable-sized "inode" representations for different types of files amortizes the cost across all subsequent operations on a file. It remains to be verified whether this is proved to be valid on a broad set of workloads and whether the same argument would apply in the context of adding flexibility to existing filesystems without requiring invasive changes.

Would the reliance on C++ in KFS be concern for application to existing Linux filesystems?

Inheritance has proven to be very useful during the development of per-element specialization in KFS, as it simplified coding to a great extent and enabled workload-specific optimizations. However, being able to move to C with a minimalist scheme may be a desirable goal when working with existing filesystems in the linux kernel.

4.4 Ability to drop experimental layout changes easily

As described in section 2.3, the problem of being stuck with on-disk format changes once made necessitates a stringent approach towards adoption of layout improvements which may result in years of lead time into actual deployment of state-of-the-art filesystem enhancements.

In KFS, as we add new formats, we can still work with old ones for compatibility, but the old ones can be kept out of the mainstream implementation. The code is activated when reading old representations, and we can on the fly "migrate" to the new format as we access it, albeit at a certain run-time overhead.

This does not however address the issue of handling backward compatibility with older kernels. Perhaps it would make sense to include a compatibility information scheme at a per-PSO level, similar to the ext2/3 filesystem's superblock level compatibility bitmaps. For example, in the situation where we are able to extend a given PSO type to a new scheme in a way that is backward compatible with the earlier PSO type (e.g. as in case of the directory indexing feature), we would like to indicate that so that an older kernel does not reject it.

5 Future work

Section 4 has discussed ongoing work on KFS that, in the short term, may result in useful insights for achieving an adaptable filesystem framework for Linux. In this section we discuss new work to be done that is essential to realizing this adaptable framework.

5.1 Adaptability in the filesystem checker and tools

With adaptable building blocks and support for multiple alternate formats as part of a per element specialization, it follows that file-system checker changes would be required to be able to recognize, load, and handle new PSOs as well as perform some level of general consistency checks based on the indication of location and sizes common to all elements, and parsing the record map(s). As with existing filesystem checkers, backward compatibility remains a tricky issue. The PSO compatibility scheme discussed earlier could be used to flag situations where newer versions of tools are required. Likewise, other related utilities like low-level backup utilities (e.g dump), migration tools, defragmenter, debugfs, etc would need to be dynamically extendable to handle new formats.

5.2 Address the problem of sharing granular building blocks across filesystems

KFS was not originally designed with the intent of enabling building-block sharing across existing file systems. However, given potential benefits in factoring core ext2/3 code, data-consistency schemes and additional capabilities (e.g. clustering), as well as extensions to libfs beyond its current limited scope of application, it is natural to ask whether KFS-type constructs could be useful in this context.

Could the same principles that allow per element flexibility through building block composition be taken further to enable abstraction of these building blocks in a way that not tightly tied to the containing filesystem? Design issues that may need to be explored in order to evaluate the viability of this possibility include figuring out how a combination of PSOs, e.g. alternate layouts and alternate consistency, schemes could be built efficiently in the context of an existing filesystem in a manner that is reusable in the context of another filesystem. The effort involved in trying to refactor existing code into PSOs may not be small; a better approach may be to start this with a few simple building blocks and use the framework for new building blocks created from here on.

5.3 Additional Issues

Another aspect that needs further exploration is whether the inclusion of building blocks and associated specialization adds to overall testing complexity for distributions, or if the framework can be enhanced to enable a systematic approach to be devised to simplify such verification.

6 Conclusions

We presented KFS and discussed that the experience so far indicates that KFS can be a powerful approach to support flexibility of services in a file system down to a per-element granularity. We also argued that the approach does not come with unacceptable performance overheads, and that it allows for workload-specific optimizations. We believe that the flexibility in KFS makes it a good prototype environment for experimenting with new file system resource management policies or file representations. As new emerging workloads appear, KFS can be useful to the Linux community by providing evaluation results for alternative representations and by advancing the application of different data layouts for different files within the same filesystem, determined either statically or dynamically in response to changing access patterns.

In this paper we propose a new exploration of KFS: to investigate how its building-block approach could be abstracted from KFS's implementation to allow code sharing among file systems, providing a library-like collection of alternative implementations to be experimented with across file systems. We do not have yet evidence that this approach is feasible, but as we improve the integration of KFS (originally designed for the K42 operating system) with Linux 2.6 we hope to have a better understanding of KFS's general applicability.

Acknowledgements

We would like to thank Dave Kleikamp for contributing the section on JFS flexibility, and Christoph Hellwig for his inputs on XFS.

Orran Krieger started KFS based on his experience with designing and implementing HFS. He has been a steady source of guidance and motivation for KFS developers, always available for in-depth design reviews that improved KFS and made our work more fun. We thank him for his insight and support.

We would like to thank Paul McKenney, Val Henson, Mingming Cao, and Chris Mason for their valuable review feedback on early drafts of this paper, and H. Peter Anvin and Mel Gorman for helping us improve our proposal. This paper might never have been written but for insightful discussions with Stephen Tweedie, Theodore T'so, and Andreas Dilger over the course of the last year. We would like to thank them and other ext3 filesystem developers for their steady focus on continuously improving the ext3 filesystem, devising interesting techniques to address conflicting goals of dependability vs state-of-the-art advancement. Finally we would like to thank the linux-fsdevel community for numerous discussions over years which motivated this paper.

This work was partially supported by DARPA under contract NBCH30390004.

Availability

KFS is released as open source as part of the K42 system, available from a public CVS repository; for details refer to the K42 web site: `http://www.research.ibm.com/K42/`.

Legal Statement

Copyright © 2006 IBM.

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM and OS/2 are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or service marks of others.

References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document is provied "AS IS," with no express or implied warranties. Use the information in this document at your own risk.

References

[1] Ray Bryant, Ruth Forester, and John Hawkes. Filesystem performance and scalability in linux 2.4.17. In *USENIX Annual Technical Conference*, 2002.

[2] M. Cao, T.Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the art: Where we are with the ext3 filesystem. In *Proceedings of the Ottawa Linux Symposium(OLS)*, pages 69–96, 2005.

[3] Jonathan Corbet. Which filesystem for samba4? `http://lwn.net/Articles/112566/`.

[4] Jonathan Corbet. Creating linux virtual file systems. `http://lwn.net/Articles/57369/`, November 2003.

[5] Jonathan Corbet. The OCFS2 filesystem. `http://lwn.net/Articles/137278/`, May 2005.

[6] Alan Cox. Posting on linux-fsdevel. `http://marc.theaimsgroup.com/?l=linux-fsdevel&m=112558745427067&w=2`, September 2005.

[7] Dilma da Silva, Livio Soares, and Orran Krieger. KFS: Exploring flexilibity in file system design. In *Proc. of the Brazilian Workshop in Operating Systems*, Salvador, Brazil, August 2004.

[8] Greg Ganger and Frans Kaashoek. Embedded inodes and explicit gruopings: Exploiting disk bandwith for small files. In *Proceedings of the 1997 Usenix Annual Technical Conference*, pages 1–17, January 1997.

[9] Val Henson, Zach Brown, Theodore Ts'o, and Arjan van de Ven. Reducing fsck time for ext2 filesystems. In *Proceedings of the Ottawa Linux Symposium(OLS)*, 2006.

[10] The K42 operating system, `http://www.research.ibm.com/K42/`.

[11] Kfs performance experiments. `http://k42.ozlabs.org/Wiki/KfsExperiments`, 2006.

[12] O. Krieger and M. Stumm. HFS: A performance-oriented flexible filesystem based on build-block compositions. *ACM Transactions on Computer Systems*, 15(3):286–321, 1997.

[13] Orran Krieger. *HFS: A Flexible File System for Shared-Memory Multiprocessors*. PhD thesis, Department of Electrical and Computer Engineering, University of Toronto, 1994.

[14] Orran Krieger and Michael Stumm. HFS: A flexible file system for large-scale multiprocessors. In *Proceedings of the DAGS/PC Symposium (The Second Annual Dartmouth Institute on Advanced Graduate Studies in Parallel Computation)*, 1993.

[15] Namesys. Reiser4. `http://www.namesys.com/v4/v4.html`, August 2004.

[16] Daniel Phillips. A directory index for ext2. In *5th Annual Linux Showcase and Conference*, pages 173–182, 2001.

[17] Justin Piszcz. Benchmarking File Systems Part II.

[18] Livio Soares, Orran Krieger, and Dilma Da Silva. Meta-data snapshotting: A simple mechanism for file system consistency. In *SNAPI'03 (International Workshop on Storage Network Architecture and Parallel I/O)*, pages 41–52, 2003.

[19] John Troy Stepan. Linux File Systems Comparative Performance. *Linux Gazette*, January 2006. `http://linuxgazette.net/122/TWDT.html`.

[20] IBM JFS Core Team.

[21] SGI XFS Team. XFS: a high-performance journaling filesystem.

[22] Stephen Tweedie and Theodore Y Ts'o. Planned extensions to the linux ext2/3 filesystem. In *USENIX Annual Technical Conference*, pages 235–244, 2002.

[23] Stephen C Tweedie. Re: Rfc: mke2fs with dir_index, resize_inode by default, March, 2006.

[24] Linux kernel sourcecode vfs documentation. file `Documentaion/filesystems/vfs.txt`.

Multiple Instances of the Global Linux Namespaces

Eric W. Biederman
Linux Networx
ebiederman@lnxi.com

Abstract

Currently Linux has the filesystem namespace for mounts which is beginning to prove useful. By adding additional namespaces for process ids, SYS V IPC, the network stack, user ids, and probably others we can, at a trivial cost, extend the UNIX concept and make novel uses of Linux possible. Multiple instances of a namespace simply means that you can have two things with the same name.

For servers the power of computers is growing, and it has become possible for a single server to easily fulfill the tasks of what previously required multiple servers. Hypervisor solutions like Xen are nice but they impose a performance penalty and they do not easily allow resources to be shared between multiple servers.

For clusters application migration and preemption are interesting cases but almost impossibly hard because you cannot restart the application once you have moved it to a new machine, as usually there are resource name conflicts.

For users certain desktop applications interface with the outside world and are large and hard to secure. It would be nice if those applications could be run on their own little world to limit what a security breach could compromise.

Several implementations of this basic idea have been done succsessfully. Now the work is to create a clean implementation that can be merged into the Linux kernel. The discussion has begun on the *linux-kernel* list and things are slowly progressing.

1 Introduction

1.1 High Performance Computing

I have been working with high performance clusters for several years and the situation is painful. Each Linux box in the cluster is refered to as a node, and applications running or queued to run on a cluster are jobs.

Jobs are run by a batch scheduler and, once launched, each job runs to completion typically consuming 99% of the resources on the nodes it is running on.

In practice a job cannot be suspended, swapped, or even moved to a different set of nodes once it is launched. This is the oldest and most primitive way of running a computer. Given the long runs and high computation overhead of HPC jobs it isn't a bad fit for HPC environments, but it isn't a really good fit either.

Linux has much more modern facilities. What prevents us from just using them?

HPC jobs currently can be suspended, but that just takes them off the cpu. If you have sufficient swap there is a chance the jobs can even be pushed to swap but frequently these application lock pages in memory so they can be ensured of low latency communication.

The key problem is simply having multiple machines and multiple kernels. In general, how to take an application running under one Linux kernel and move it completely to another kernel is an unsolved problem.

The problem is unsolved not because it is fundamentally hard, but simply because it has not be a problem in the UNIX environment. Most applications don't need multiple machines or even big machines to run on (especially with Moore's law exponentially increasing the power of small machines). For many of the rest the large multiprocessor systems have been large enough.

What has changed is the economic observation that a cluster of small commodity machines is much cheaper and equally as fast as a large supercomputer.

The other reason this problem has not been solved (besides the fact that most people working on it are researchers) is that it is not immediately obvious what a general solution is. Nothing quite like it has been done before so you can't look into a text book or into the archives of history and know a solution. Which in the broad strokes of operating system theory is a rarity.

The hard part of the problem also does not lie in the obvious place people first look— how to save all of the state of an application. Instead, the problem is how do you restore a saved application so it runs successfully on another machine.

The problem with restoring a saved application is all of the global resources an application uses. Process ids, SYS V IPC identifiers, filenames, and the like. When you restore an application on another machine there is no guarantee that it can reuse the same global identifiers as another process on that machine may be using those identifiers.

There are two general approaches to solving this problem. Modifying things so these global machine identifiers are unique across all machines in a cluster, or modifying things so these machine global identifiers can be repeated on a single machine. Many attempts have been made to scale global identifiers cluster-wide—Mosix, OpenSSI, bproc, to name a few—and all of them have had to work hard to scale. So I choose to go with an implementation that will easily scale to the largest of clusters, with no communication needed to do so.

This has the added advantage that in a cluster it doesn't change the programming model presented to the user. Just some machines will now appear as multiple machines. As the rise of the internet has shown building applications that utilize multiple machines is not foreign to the rest of the computing world either.

To make this happen I need to solve the challenging problem of how to refactor the UNIX/Linux API so that we can have multiple instances of the global Linux namespaces. The Plan 9 inspired mount/filesystem namespace has already proved how this can be done and is slowly proving useful.

1.2 Jails

Outside the realm of high perfomance computing people have been restricting their server application to chroot jails for years. The problems with chroot jails have become well understood and people have begun fixing them. First BSD jails, and then Solaris containers are some of the better known examples.

Under Linux the open source community has not been idle. There is the linux-jail project, Vserver, Openvz, and related projects like SELinux, UML, and Xen.

Jails are a powerful general purpose tool useful for a great variety of things. In resource utilization jails are cheap, dynamically loading glibc is likely to consume more memory than the additional kernel state needed to track a jail. Jails allow applications to be run in simple stripped down environments, increasing security, and decreasing maintenance costs, while leaving system administrators with the familiar UNIX environment.

The only problem with the current general purpose implementation of jails under Linux is nothing has been merged into the mainline kernel, and the code from the various projects is not really mergable as it stands. The closest I have seen is the code from the linux-jail project, and that is simply because it is less general purpose and implemented completely as a Linux security module.

Allowing multiple instances of global names which is needed to restore a migrated application is a more constrained problem than that of simply implementing a jail. But a jail that ensures you can have multiple instances of all of the global names is a powerful general purpose jail that you can run just about anything in. So the two problems can share a common kernel solution.

1.3 Future Directions

A cheap and always available jail mechanism is also potentially quite useful outside the realm of high performance computing and server applications. A general purupose checkpoint/restart mechanism can allow desktop users to preserve all of their running applications when they log out. Vulnerable or untrusted applications like a web browser or an irc client could be contained so that if they are attacked all that is gained is the ability to browse the web and draw pictures in an X window.

There would finally be answer to the age old question: How do I preserve my user space while upgrading my kernel?

All this takes is an eye to designing the interfaces so they are general purpose and nestable. It should be possible to have a jail inside a jail inside a jail forever, or at least until there don't exist the resources to support it.

2 Namespaces

How much work is this? Looking at the existing patches it appears that 10,000 to 20,000 lines of code will ultimately need to be touched. The core of Linux is about 130,000 lines of code, so we will need to touch between 7% and 15% of the core kernel code. Which clearly indicates that one giant patch to do everything even if it was perfect would be rejected simply because it is too large to be reviewed.

In Linux there are multiple classes of global identifiers (i.e. process id, SYS V IPC keys, user ids). Each class of identifier can be thought of living in its own namespace.

This gives us a natural decomposition to the problem, allowing each namespace to be modified separately so we can support multiple instances of that namespace. Unfortunately this also increases the difficulty of the problem, as we need to modify the kernel's reporting and configuration interfaces to support multiple instances of a namespace instead of having them tightly coupled.

The plan then is simple. Maintain backwards compatibility. Concentrate on one namespace at a time. Focus on implementing the ability to have multiple objects of a given type, with the same name. Configure a namespace from the inside using the existing interfaces. Think of these things ultimately not as servers or virtual machines but as processes with peculiar attributes. As far as possible implement the namespaces so that an application can be given the capability bit for that allows full control over a namespace and still not be able to escape. Think in terms of a recursive implementation so we always keep in mind what it takes to recurse indefinitely.

What the system call interface will be to create a new instance of a namespace is still up for debate. The current contenders are a new CLONE_ flag or individual system calls. I personally think a flag to clone and unshare is all that is needed but if arguments are actually needed a new system call makes sense.

Currently I have identified ten separate namespaces in the kernel. The filesystem mount namespace, uts namespace, the SYS V IPC namespace, network namespace, the pid namespace, the uid namespace, the security namespace, the security keys namespace, the device namespace, and the time namespace.

2.1 The Filesystem Mount Namespace

Multiple instances of the filesystem mount namespace are already implemented in the stable Linux kernels so there are few real issues with implementing it. There are still outstanding question on how to make this namespace usable and useful to unpriviledged processes, as well as some ongoing work to allow the filesystem mount namespace to allow bind mounts to have bind flags. For example, so the the bind mount can be restricted read only when other mounts of the filesystem are still read/write.

```
int uname(struct utsname *buf);
struct utsname {
        char sysname[];
        char nodename[];
        char release[];
        char version[];
        char machine[];
        char domainname[];
};
```

Figure 1: uname

CAP_SYS_ADMIN is currently required to modify the mount namespace, although there has been some discussion on how to relax the restrictions for bind mounts.

2.2 The UTS Namespace

The UTS namespace characterizes and identifies the system that applications are running on. It is characterizeed by the uname system call. uname returns six strings describing the current system. See Figure 1.

The returned utsname structure has only two members that vary at runtime nodename and domainname. nodename is the classic hostname of a system. domainname is the NIS domainname. CAP_SYS_ADMIN is required to change these values, and when the system boots they start out as "(none)".

The pieces of the kernel that report and modify the utsname structure are not connected to any of the big kernel subsystems, build even when CONFIG_NET is disabled, and use CAP_SYS_ADMIN instead of one of the more specific capabilities. This clearly shows that the code has no affiliation with one of the larger namespaces.

Allowing for multiple instances of the UTS namespace is a simple matter of allocating a new copy of `struct utsname` in the kernel for each different instance of this namespace, and modifying the system calls that modify this value to lookup the appropriate instance of `struct utsname` by looking at `current`.

2.3 The IPC Namespace

The SYS V interprocess communication namespace controls access to a flavor of shared memory, semaphores, message queues introduced in SYS V UNIX. Each object has associated with it a `key` and an `id`, and all objects are globally visible and exist until they are explicitly destroyed.

The `id` values are unique for every object of that type and assigned by the system when the object is created.

The `key` is assigned by the application usually at compile time and is unique unless it is specified as `IPC_PRIVATE`. In which case the `key` is simply ignored.

The ipc namespace is currently limited by the following universally readable and uid 0 setable `sysctl` values:

- `kernel.shmmax` The maximum shared memory segment size.
- `kernel.shmall` The maximum combined size of all shared memory segments.
- `kernel.shmni` The maximum number of shared memory segments.
- `kernel.msgmax` The maximum message size.
- `kernel.msgmni` The maximum number of message queues.
- `kernel.msgmnb` The maximum number of bytes in a message queue.
- `kernel.sem` An array of 4 control integers
  - `sc_semmsl` The maximum number of semaphores in a semaphore set.
  - `sc_semmns` The maximum number of semaphores in the system.
  - `sc_semopm` The maximum number of semaphore operations in a single system call.
  - `sc_semmni` The maximum number of semaphore sets in the system.

Operations in the ipc namespace are limited by the following capabilities:

- `CAP_IPC_OWNER` Allows overriding the standard ipc ownership checks, for the following operations: `shm_attach`, `shm_stat`, `shm_get`, `msgrcv`, `msgsnd`, `msg_stat`, `msg_get`, `semtimedop`, `sem_getall`, `sem_setall`, `sem_stat`, `sem_getval`, `sem_getpid`, `sem_getncnt`, `sem_getzcnt`, `sem_setval`, `sem_get`.

  For filesystems `namei.c` uses `CAP_DAC_OVERRIDE`, and `CAP_DAC_READ_SEARCH` to provide the same level of control.
- `CAP_IPC_LOCK` Required to control locking of shared memory segments in memory.
- `CAP_SYS_RESOURCE` Allows setting the maximum number of bytes in a message queue to exceed `kernel.msgmnb`
- `CAP_SYS_ADMIN` Allows changing the ownership and removing any ipc object.

Allowing for multiple instances of the ipc namespace is a straightforward process of duplicating the tables used for lookup by `key` and id, and modifying the code to use current to select the appropriate table. In addition the sysctls need to be modified to look at current and act on the corresponding copy of the namespace.

The ugly side of working with this namespace is the capability situation. CAP_IPC_OWNER trivially becomes restricted to the current ipc namespace. CAP_IPC_LOCK still remains dangerous. CAP_DAC_OVERRIDE and CAP_DAC_READ_SEARCH might be ok, it really depends on the state of the filesystem mount namespace. CAP_SYS_RESOURCE and CAP_SYS_ADMIN are still unsafe to give to untrusted applications.

2.4 The Network Namespace

By volume the code implementing the network stack is the largest of the subsystems that needs its own namespace. Once you look at the network subsystem from the proper slant it is straightforward to allow user space to have what appears to be multiple instances of the network stack, and thus you have a network namespace.

The core abstractions used by the network stack are processes, sockets, network devices, and packets. The rest of the network stack is defined in terms of these.

In adding the network namespace I add a few simple rules.

- A network device belongs to exactly one network namespace.
- A socket belongs to exactly one network namespace.

A packet comes into the network stack from the outside world through a network device. We can the look at that device to find the network namespace and the rules to process that packet by.

We generate a packet and feed it to the kernel through a socket. The kernel looks at the socket, finds the network namespace, and from there the rules to process the packet by.

What this means is that most of the network stack global variables need to be moved into the network namespace data structure. Hopefully we can write this so the extra level of indirection will not reduce the performance of the network stack.

Looking up the network stack global variables through the network namespace is not quite enough. Each instance of the network namespace needs its own copy of the loopback device, an interface needs to be added to move network devices between network namespaces, and a two headed tunnel device (a cousin of the loopback device) needs to be added so we can send packets between different network namespaces.

With these in place it is safe to give processes a separate network namespace: CAP_NET_BIND_SERVICE, CAP_NET_BROADCAST, CAP_NET_ADMIN, and CAP_NET_RAW. All of the functionality will work and working through the existing network devices there won't be an ability to escape the network namespace. Care should be given to giving an untrusted process access to real network devices, though, as hardware or software bugs in the implementation of that network device could be reduce the security.

The future direction of the network stack is towards Van Jackobson network channels, where more of the work is pushed towards process context, and happening in sockets. That work appears to be a win for network namespaces in two ways. More work happening in process context and in well defined sockets means it is easier to lookup the network namespace, and thus cheaper. Having a lightweight packet classifier in the network drivers should allow a single network device to appear to user space as multiple network devices each with a different hardware address. Then based upon the destination hardware address the packet can be placed in one of several different network namespaces. Today to get the same semantics I need to configure the primary network device as a router, or configure ethernet bridging between the real network and the network interface that sends packets to the secondary network namespace.

2.5 The Process Id Namespace

The venerable process id is usually limited to 16bits so that the bitmap allocator is efficient and so that people can read and remember the ids. The identifiers are allocated by the kernel, and identifiers that are no longer in use are periodically reused for new processes. A single identifier value can refer to a process, a thread group, a process group and to a session, but every use starts life as a process identifier.

The only capability associated with process ids is `CAP_KILL` which allows sending signals to any process even if the normal security checks would not allow it.

Process identifiers are used to identify the current process, to identify the process that died, to specify a process or set of processes to send a signal to, to specify a process or set of processes to modify, to specify a process to debug, and in system monitoring tools to specify which process the information pertains to. Or in other words process identifiers are deeply entrenched in the user/kernel interface and are used for just about everything.

In a lot of ways implementing a process id namespace is straightforward as it is clear how everything should look from the inside. There should be a group of all of the processes in the namespace that `kill -1` sends signals to. Either a new pid hash table needs to be allocated or the key in the pid hash table needs to be modified to include the pid namespace. A new pid allocation bitmap needs to be allocated. `/proc/sys/pid_max` needs to be modified to refer to the current pid allocation bitmap. When the pid namespace exits all of the processes in the pid namespace need to be killed. Kernel threads need to be modified to never start up in anything except the default pid namespace. A process that has pid 1 must exist that will not receive any signals except for the ones it installs a signal handler for.

How a pid namespace should look from the outside is a much more delicate question. How should processes in a non default pid namespace be displayed in `/proc`? Should any of the process in a pid namespace show up in any other pid namespace? How much of the existing infrastructure that takes pids should continue to work?

This is one of the few areas where the discussion on the kernel list has come to a complete standstill, as an inexpensive technical solution to everyones requirements was not visible at the time of the conversation.

A big piece of what makes the process id namespace different is that processes are organized into a hierarchical tree. Maintaining the parent/child relationship between the process that initiates the pid namespace and the first process in the new pid namespace requires first process in the new pid namespace have two pids. A pid in the namespace of the parent and pid 1 in its own namespace. This results in namespaces that are hierarchical unlike most namespaces that are completely disjoint.

Having one process with two pids looks like a serious problem. It gets worse if we want that process to show up in other pid namespaces.

After looking deeply at the underlying mechanisms in the kernel I have started moving things away from `pid_t` to pointers to `struct pid`. The immediate gain is that the kernel becomes protected from pid wraparound issues.

Once all of the references that matter are `struct pid` pointers inside the kernel a different implementation becomes possible. We can hang multiple <pid namespace, `pid_t`> tuples off `struct pid` allowing us to have a different name for the same pid in several pid namespaces.

With processes in subordinate pid namespaces at least potentially showing up when we need them we can preserve the existing UNIX api for all functions that take pids and not need to reinvent pid namespace specific solutions.

The question yet to be answered in my mind is do we always map a process's `struct pid` into all of its ancestor's pid namespaces, or do we provide a mechanism that performs those mappings on demand?

2.6 The User and Group ID Namespace

In the kernel user ids are used for both accounting and for for performing security checks. The per user accounting is connected to the `user_struct`. Security checks are done against `uid`, `euid`, `suid`, `fsuid`, `gid`, `egid`, `sgid`, `fsgid`, processes capabilities, and variables maintained by a Linux security module. The kernel allows any of the uid/gid values to rotate between slots, or, if a process has `CAP_SETUID`, arbitrary values to be set into the filesystem uids.

With a uid namespace the security checks for equality of uids become checks to ensure the entire tuple <uid namespace, uid> is equal. Which means if two uids are in different namespaces the check will always fail. So the only permitted cases across uid namespaces will be when everyone is allowed to perform the action the process is trying to perform or when the process has the appropriate capability to perform the action on any process.

An alternative in some cases to modifying all of the checks to be against <namespace, uid> tuples is to modify some of the checks to be against `user_struct` pointers.

Since uid namespaces are not persistent, mapping of a uid namespace to filesystems requires some new mechanisms. The primary mechanism is to associate with the each `super_block` the uid namespace of the filesystem; probably moving that information into each `struct inode` in the kernel for speed and flexibility.

To allow sharing of filesystem mounts between different uid namespaces requires either using acls to tag inodes with non-default filesystem namespace information or using the key infrastructure to provide a mapping between different uid namespaces.

Virtual filesystems require special care as frequently they allow access to all kinds of special kernel functionality without any capability checks if the uid of a process equals 0. So virtual filesystems like proc and sysfs must specify the default kernel uid namespace in their `superblock` or it will be trivial to violate the kernel security checks.

There is a question of whether the change in rules and mechanisms should take place in the core kernel code, making it uid namespace aware, or in a Linux security module. A key of that decision is the uid hash table and `user_struct`. From my reading of the kernel code it appears that current Linux security modules can only further restrict the default kernel permissions checks and there is not a hook that makes it possible to allocate a different `user_struct` depending on security module policies.

Which means at least the allocation of `user_struct`, and quite possibly making all of the uid checks fail if the uid namespaces are not equal, should happen in the core of the kernel with security modules standing in the background providing really advanced facilities.

With a uid namespace it becomes safe to give untrusted users `CAP_SETUID` without reducing security.

2.7 Security Modules and Namespaces

There are two basic approaches that can be pursued to implement multiple instances of user space. Objects in the kernel can be isolated by policy and security checks with security modules, or they can be isolated by making visible only the objects you are allowed to access by using namespaces.

The Linux Jail module (`http://sf.net/projects/linuxjail`) implemented by "Serge E. Hallyn" <serue@us.ibm.com> is a good example of what can be done with just a security module and isolating a group of processes with permission checks and policy rather than simply making the inaccessible parts of the system disappear.

Following that general principle Linux security modules have two different roles they can play when implementing multiple instances of user space. They can make up for any unimplemented kernel namespace by isolating objects with additional permission checks, which is good as a short term solution. Linux security modules modified to be container aware can also provide for enhanced security enforcement mechanisms in containers. In essence this second modification is the implementation of a namespace for security mechanisms and policy.

2.8 The Security Keys Namespace

Not long ago someone added to the kernel what is the frustration of anyone attempting to implementing namespaces to allow for the migration of user space. Another obscure and little known global namespace.

In this case each key on a key ring is assigned a global `key_serial_t` value. `CAP_SYS_ADMIN` is used to guard ownership and permission changes.

I have yet to look in detail but at first glance this looks like one of the easy cases, where we can just simply implement another copy of the lookup table. It appears the `key_serial_t` values are just used for manipulation of the security keys, from user space.

2.9 The Device Namespace

Not giving children in containers `CAP_SYS_MKNOD` and not mounting sysfs is sufficient to prevent them from accessing any device nodes that have not been audited for use by that container. Getting a new instance of the uid/gid namespace is enough to remove access from magic sysfs entries controlling devices although there is some question on how to bring them back.

For purposes of migration, unless all devices a set of processes has access to are purely virtual, pretending the devices haven't changed is nonsense. Instead it makes much more sense to explicitly acknowledge the devices have changed and send hotplug remove and add events to the set of processes.

With the use of hotplug events the assumption that the global major and minor numbers that a device uses are constant is removed.

Equally as sensitive as CAP_SYS_MKNOD, and probably more important if mounting sysfs is allowed, is CAP_CHOWN. It allows changing the owner of a file. Since it would be required to change the sysfs owner before a sensitive file could be accessed.

So in practice managing the device namespace appears to be a user space problem with restrictions on CAP_SYS_MKNOD and CAP_CHOWN being used to implement the filter policy of which devices a process has access to.

2.10 The Time Namespace

The kernel provides access to several clocks CLOCK_REALTIME, CLOCK_MONOTONIC, CLOCK_PROCESS_CPUTIME_ID, and CLOCK_THREAD_CPUTIME_ID being the primaries.

CLOCK_REALTIME reports the current wall clock time.

CLOCK_MONOTONIC is similar to CLOCK_REALTIME except it cannot be set and thus never backs up.

CLOCK_PROCESS_CPUTIME_ID reports how much aggregate cpu time the threads in a process have consumed.

CLOCK_THREAD_CPUTIME_ID reports how much cpu time an individual thread has consumed.

If process migration is not a concern none of these clocks except possibly CLOCK_REALTIME is interesting. In the context of process migration all of these clocks become interesting.

The thread and process clocks simply need an offset field so the amount of time spent on the previous machine can be added in. So that we can prevent the clocks from going backwards.

The monotonic timer needs an offset field so that we can guarantee that it never goes backwards in the presence of process migration.

The realtime clock matters the least but having an additional offset field for clock adds additional flexibility to the system and comes at practically no cost.

All of the clocks except for CLOCK_MONOTONIC support setting the clock with clock_settime so the existing control interfaces are sufficient. For the monotonic clock things are different, clock_settime is not allowed to set the clock, ensuring that the time will never run backwards, and there is a huge amount of sense in that logic.

The only reasonable course I can see is setting the monotonic clock when the time namespace is created. Which probably means we will need a syscall (and not a clone flag) to create the clone flag and we should provide it with minimum acceptable value of the monotonic clock.

3 Modifications of Kernel Interfaces

One of the biggest challenges with implementing multiple namespaces is how do we modify the existing kernel interfaces in a way that retains backwards compatibility for existing applications while still allowing the reality of the new situation to be seen and worked with.

Modifying existing system calls is probably the easiest case. Instead of reference global variables we reference variables through `current`.

Modifying `/proc` is trickier. Ideally we would introduce a new subdirectory for each class of namespace, and in that folder list each instance of that namespace, adding symbolic links from the existing names where appropriate. Unfortunately it is not possible to obtain a list of namespaces and the extra maintenance cost does not yet seem to just the extra complexity and cost of a linked list. So for proc what we are likely to see is the information for each namespace listed in `/proc/pid` with a symbolic link from the existing location into the new location under `/proc/self/`.

Modifying `sysctl` is fairly straightforward but a little tricky to implement. The problem is that `sysctl` assumes it is always dealing with global variables. As we put those variables into namespaces we can no longer store the pointer into a global variable. So we need to modify the implementation of `sysctl` to call a function which takes a `task_struct` argument to find where the variable is located. Once that is done we can move `/proc/sys` into `/proc/pid/sys`.

Modifying `sysfs` doesn't come up for most of the namespaces but it is a serious issue for the network namespace. I haven't a clue what the final outcome will be but assuming we want global visibility for monitor applications something like `/proc/pid` and `/proc/self` needs to be added so we can list multiple instances and add symbolic links from their old location in `sysfs`.

The `netlink` interface is the most difficult kernel interface to work with. Because control and query packets are queued and not necessarily processed by the application that sends the query, getting the context information necessary to lookup up the appropriate global variables is a challenge. It can even be difficult to figure out which port namespace to reply to. As long as the only users of netlink are part of the networking stack I have an implementation that solves the problems. However, as netlink has been suggested for other things, I can't count on just the network stack processing packets.

Resource counters are one of the more challenging interfaces to specify. Deciding if an interface should be per namespace or global is a challenge, and answering the question how does this work when we have recursive instances of a namespace. All of these concerns are exemplified when there are multiple untrusted users on the system. For starters we should be able to punt and implement something simple and require `CAP_SYS_RESOURCE` if there are any per namespace resource limits. Which should leave us with a simple and correct implementation. Then the additional concerns can be addressed from there.

Fully Automated Testing of the Linux Kernel

Martin Bligh
Google Inc.
`mbligh@mbligh.org`

Andy P. Whitcroft
IBM Corp.
`andyw@uk.ibm.com`

Abstract

Some changes in the 2.6 development process have made fully automated testing vital to the ongoing stability of Linux®. The pace of development is constantly increasing, with a rate of change that dwarfs most projects. The lack of a separate 2.7 development kernel means that we are feeding change more quickly and directly into the main stable tree. Moreover, the breadth of hardware types that people are running Linux on is staggering. Therefore it is vital that we catch at least a subset of introduced bugs earlier on in the development cycle, and keep up the quality of the 2.6 kernel tree.

Given a fully automated test system, we can run a broad spectrum of tests with high frequency, and find problems soon after they are introduced; this means that the issue is still fresh in the developers mind, and the offending patch is much more easily removed (not buried under thousands of dependant changes). This paper will present an overview of the current early testing publication system used on the `http://test.kernel.org` website. We then use our experiences with that system to define requirements for a second generation fully automated testing system.

Such a system will allow us to compile hundreds of different configuration files on every release, cross-compiling for multiple different architectures. We can also identify performance regressions and trends, adding statistical analysis. A broad spectrum of tests are necessary—boot testing, regression, function, performance, and stress testing; from disk intensive to compute intensive to network intensive loads. A fully automated test harness also empowers other other techniques that are impractical when testing manually, in order to make debugging and problem identification easier. These include automated binary chop search amongst thousands of patches to weed out dysfunctional changes.

In order to run all of these tests, and collate the results from multiple contributors, we need an open-source client test harness to enable sharing of tests. We also need a consistent output format in order to allow the results to be collated, analysed and fed back to the community effectively, and we need the ability to "pass" the reproduction of issues from test harness to the developer. This paper will describe the requirements for such a test client, and the new open-source test harness, Autotest, that we believe will address these requirements.

1 Introduction

It is critical for any project to maintain a high level of software quality, and consistent interfaces to other software that it uses or uses it.

There are several methods for increasing quality, but none of these works in isolation, we need a combination of:

- skilled developers carefully developing high quality code,
- static code analysis,
- regular and rigorous code review,
- functional tests for new features,
- regression testing,
- performance testing, and
- stress testing.

Whilst testing will never catch all bugs, it will improve the overall quality of the finished product. Improved code quality results in a better experience not only for users, but also for developers, allowing them to focus on their own code. Even simple compile errors hinder developers.

In this paper we will look at the problem of automated testing, the current state of it, and our views for its future. Then we will take a case study of the test.kernel.org automated test system. We will examine a key test component, the client harness, in more detail, and describe the Autotest test harness project. Finally we will conclude with our vision of the future and a summary.

2 Automated Testing

It is obvious that testing is critical, what is perhaps not so obvious is the utility of regular testing at all stages of development. It is important to catch bugs as soon as possible after they are created as:

- it prevents replication of the bad code into other code bases,
- fewer users are exposed to the bug,
- the code is still fresh in the authors mind,
- the change is less likely to interact with subsequent changes, and
- the code is easy to remove should that be required.

In a perfect world all contributions would be widely tested before being applied; however, as most developers do not have access to a large range of hardware this is impractical. More reasonably we want to ensure that any code change is tested before being introduced into the mainline tree, and fixed or removed before most people will ever see it. In the case of Linux, Andrew Morton's `-mm` tree (the de facto development tree) and other subsystem specific trees are good testing grounds for this purpose.

Test early, test often!

The open source development model and Linux in particular introduces some particular challenges. Open-source projects generally suffer from the lack of a mandate to test submissions and the fact that there is no easy funding model for regular testing. Linux is particularly hard hit as it has a constantly high rate of change, compounded with the staggering diversity of the hardware on which it runs. It is completely infeasible to do this kind of testing without extensive automation.

There is hope; machine-power is significantly cheaper than man-power in the general case. Given a large quantity of testers with diverse hardware it should be possible to cover a useful subset of the possible combinations. Linux as a project has plenty of people and hardware; what is needed is a framework to coordinate this effort.

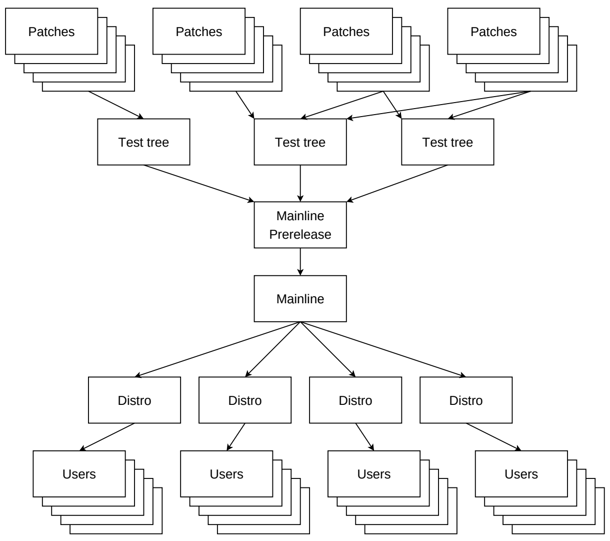


Figure 1: Linux Kernel Change Flow

2.1 The Testing Problem

As we can see from the diagram in figure 1 Linux's development model forms an hourglass starting highly distributed, with contributions being concentrated in maintainer trees before merging into the development releases (the `-mm` tree) and then into mainline itself. It is vital to catch problems here in the neck of the hourglass, before they spread out to the distros—even once a contribution hits mainline it is has not yet reached the general user population, most of whom are running distro kernels which often lag mainline by many months.

In the Linux development model, each actual change is usually small and attribution for each change is known making it easy to track the author once a problem is identified. It is clear that the earlier in the process we can identify there is a problem, the less the impact the change will have, and the more targeted we can be in reporting and fixing the problem.

Whilst contributing untested code is discouraged we cannot expect lone developers to be able to do much more than basic functional testing, they are unlikely to have access to a wide range of systems. As a result, there is an opportunity for others to run a variety of tests on incoming changes before they are widely distributed. Where problems are identified and flagged, the community has been effective at getting the change rejected or corrected.

By making it easier to test code, we can encourage developers to run the tests before ever submitting the patch; currently such early testing is often not extensive or rigorous, where it is performed at all. Much developer effort is being wasted on bugs that are found later in the cycle when it is significantly less efficient to fix them.

2.2 The State of the Union

It is clear that a significant amount of testing resource is being applied by a variety of parties, however most of the current testing effort goes on *after* the code has forked from mainline. The distribution vendors test the code that they integrate into their releases, hardware vendors are testing alpha or beta releases of those distros with their hardware. Independent Software Vendors (ISVs) are often even later in the cycle, first testing beta or even after distro release. Whilst integration testing is always valuable, this is far too late to be doing primary testing, and makes it extremely difficult and inefficient to fix problems that are found. Moreover, neither the tests that are run, nor the results of this testing are easily shared and communicated to the wider community.

There is currently a large delay between a mainline kernel releasing and that kernel being accepted and released by the distros, embedded product companies and other derivatives of Linux. If we can improve the code quality of the mainline tree by putting more effort into testing mainline earlier, it seems reasonable to assume that those "customers" of Linux would update from the mainline tree more often. This will result in less time being wasted porting changes backwards and forwards between releases, and a more efficient and tightly integrated Linux community.

2.3 What Should we be Doing?

Linux's constant evolutionary approach to software development fits well with a wide-ranging, high-frequency regression testing regime. The "release early, release often" development philosophy provides us with a constant stream of test candidates; for example the -git snapshots which are produced twice daily, and Andrew Morton's collecting of the specialised maintainer trees into a bleeding-edge `-mm` development tree.

In an ideal world we would be regression testing at least daily snapshots of all development trees, the `-mm` tree and mainline on all possible combinations of hardware; feeding the results back to the owners of the trees and the authors of the changes. This would enable problems to be identified as early as possible in the concentration process and get the offending change updated or rejected. The test.kernel.org testing project provides a preview of what is possible, providing some limited testing of the mainline and development trees, and is discussed more fully later.

Just running the tests is not sufficient, all this does is produce large swaths of data for humans to wade through; we need to analyse the results to engender meaning, and isolate any problems identified.

Regression tests are relatively easy to analyse, they generate a clean pass or fail; however, even these can fail intermittently. Performance tests are harder to analyse, a result of 10 has no particular meaning without a baseline to compare it against. Moreover, performance tests are not 100% consistent, so taking a single sample is not sufficient, we need to capture a number of runs and do simple statistical analysis on the results in order to determine if any differences are statistically significant or not. It is also critically important to try to distinguish failures of the machine or harness from failures of the code under test.

3 Case Study: test.kernel.org

We have tried to take the first steps towards the automated testing goals we have outlined above with the testing system that generates the test.kernel.org website. Whilst it is still far from what we would like to achieve, it is a good example of what can be produced utilising time on an existing in house system sharing and testing harness and a shared results repository.

New kernel releases are picked up automatically within a few minutes of release, and a predefined set of tests are run across them by a proprietary IBM® system called ABAT, which includes a client harness called autobench. The results of these tests are then collated, and pushed to the TKO server, where they are analysed and the results published on the TKO website.

Whilst all of the test code is not currently open, the results of the testing are, which provides a valuable service to the community, indicating (at least at a gross level) a feel for the viability of that release across a range of existing machines, and the identification of some specific problems. Feedback is in the order of hours from release to results publication.

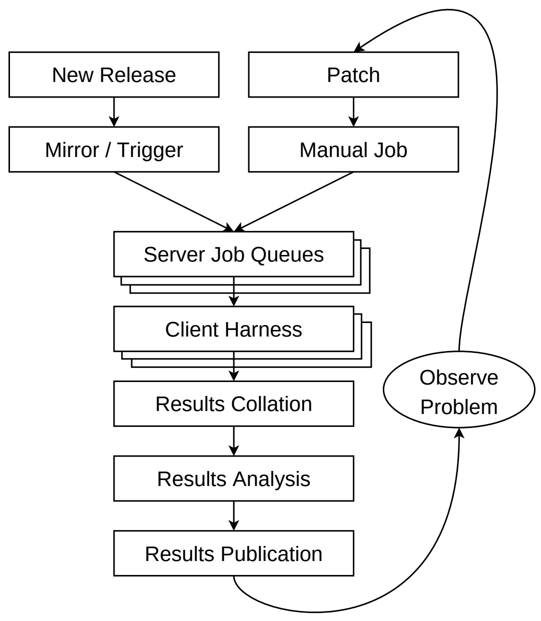


Figure 2: test.kernel.org Architecture

3.1 How it Works

The TKO system is architected as show in figure 2. Its is made up of a number of distinct parts, each described below:

The mirror / trigger engine: test execution is keyed from kernel releases; by any `-mm` tree release (2.6.16-rc1-mm1), git release (2.6.17-rc1-git10), release candidate (2.6.17-rc1), stable release (2.6.16) or stable patch release (2.6.16.1). A simple rsync local mirror is leveraged to obtain these images as soon as they are available. At the completion of the mirroring process any newly downloaded image is identified and those which represent new kernels trigger testing of that image.

Server Job Queues: for each new kernel, a predefined set of test jobs are created in the server job queues. These are interspersed with other user jobs, and are run when time is available on the test machines. IBM's ABAT server software currently fulfils this function, but a simple queueing system could serve for the needs of this project.

Client Harness: when the test system is available, the control file for that test is passed to the client harness. This is responsible for setting up the machine with appropriate kernel version, running the tests, and pushing the results to a local repository. Currently this function is served by autobench. It is here that our efforts are currently focused with the Autotest client replacement project which we will discuss in detail in section 4.4.

Results Collation: results from relevant jobs are gathered asynchronously as the tests complete and they are pushed out to test.kernel.org. A reasonably sized subset of the result data is pushed, mostly this involves stripping the kernel binaries and system information dumps.

Results Analysis: once uploaded the results analysis engine runs over all existing jobs and extracts the relevant status; this is then summarised on a per release basis to produce both overall red, amber and green status for each release/machine combination. Performance data is also analysed, in order to produce historical performance graphs for a selection of benchmarks.

Results Publication: results are made available automatically on the TKO web site. However, this is currently a "polled" model; no automatic action is taken in the face of either test failures or if performance regressions are detected, it relies on developers to monitor the site. These failures should be actively pushed back to the community via an appropriate publication mechanism (such as email, with links back to more detailed data).

Observed problems: When a problem (functional or performance) is observed by a developer monitoring the analysed and published results, this is manually communicated back to the development community (normally via email). This often results in additional patches to test, which can be manually injected into the job queues via a simple script, but currently only by an IBM engineer. These then automatically flow through with the regular releases, right through to publication on the matrix and performance graphs allowing comparison with those releases.

3.2 TKO in Action

The regular compile and boot testing frequently shakes out bugs as the patch that carried them enters the `-mm` tree. By testing multiple architectures, physical configurations, and kernel configurations we often catch untested combinations and are able to report them to the patch author. Most often these are compile failures, or boot failures, but several performance regressions have also been identified.

As a direct example, recently the performance of highly parallel workloads dropped off significantly on some types of systems, specifically with the `-mm` tree. This was clearly indicated by a drop off in the kernbench performance figures. In the graph in figure 3 we can see the sudden increase in elapsed time to a new plateau with 2.6.14-rc2-mm1. Note the vertical error bars for each data point—doing multiple test runs inside the same job allows us to calculate error margins, and clearly display them.

Once the problem was identified some further analysis narrowed the bug to a small number of scheduler patches which were then also tested; these appear as the blue line ("other" releases) in the graph. Once the regression was identified the patch owner was then contacted, several iterations of updated fixes were then produced and tested before a corrected patch was applied. This can be seen in the figures for 2.6.16-rc1-mm4.

The key thing to note here is that the regression never made it to the mainline kernel let alone into a released distro kernel; user exposure was prevented. Early testing ensured that the developer was still available and retained context on the change.

3.3 Summary

The current system is providing regular and useful testing feedback on new releases and providing ongoing trend analysis against historical releases. It is providing the results of this testing in a public framework available to all developers with a reasonable turn round time from release. It is also helping developers by testing on rarer hardware combinations to which they have no access and cannot test.

However, the system is not without its problems. The underlying tests are run on a inhouse testing framework (ABAT) which is currently not in the public domain; this prevents easy transport of these tests to other testers. As a result there is only one contributor to the result set at this time, IBM. Whilst the whole stack needs to be made open, we explain in the next section why we have chosen to start first with the client test harness.

The tests themselves are very limited, covering a subset of the kernel. They are run on a small number of machines, each with a few, fixed configurations. There are more tests which should be run but lack of developer input and lack of hardware resources on which to test prevent significant expansion.

The results analysis also does not communicate data back as effectively as it could to the community—problems (especially performance regressions) are not as clearly isolated as they could be, and notification is not as prompt and clear as it could be. More data "folding" needs to be done as we analyse across a

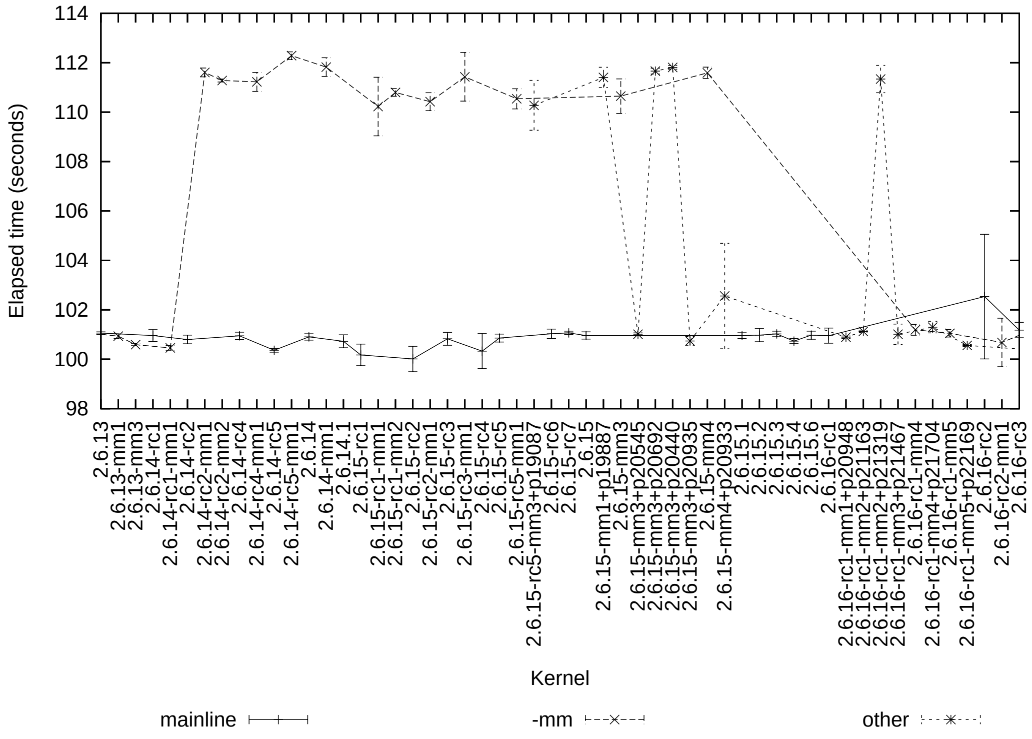


Figure 3: Kernbench Scheduler Regression

multi-dimensional space of kernel version, kernel configuration, machine type, toolchain, and tests.

4 Client Harnesses

As we have seen, any system which will provide the required level of testing needs to form a highly distributed system, and be able to run across a large test system base. This will necessitate a highly flexible client test harness; a key component of such a system. We have used our experiences with the IBM autobench client, and the TKO analysis system to define requirements for such a client. This section will discuss client harnesses in general and lead on to a discussion of the Autotest project's new test harness.

We chose to attack the problem of the client harness first as it seems to be the most pressing issue. With this solved, we can share not only results, but the tests themselves more easily, and empower a wide range of individuals and corporations to run tests easily, and share the results. By defining a consistent results format, we can enable automated collation and analysis of huge amounts of data.

4.1 Requirements / Design Goals

A viable client harness must be operable standalone or under an external scheduler infrastructure. Corporations already have significant resources invested in bespoke testing harnesses which they are not going to be willing to waste; the client needs to be able to plug into those, and timeshare resources with them. On the other hand, some testers and developers will have a single machine and want something simple they can install and use. This bimodal flexibility is particularly relevant where we want to be able to pass a failing test back to a patch author, and have them reproduce the problem.

The client harness must be modular, with a clean internal infrastructure with simple, well defined APIs. It is critical that there is clear separation between tests, and between tests and the core, such that adding a new test cannot break existing tests.

The client must be simple to use for newcomers, and yet provide a powerful syntax for complex testing if necessary. Tests across multiple machines, rebooting, loops, and parallelism all need to be supported.

We want distributed scalable maintainership, the core being maintained by a core team and the tests by the contributors. It must be able to reuse the effort that has gone into developing existing tests, by providing a simple way to encapsulate them. Whilst open tests are obviously superior, we also need to allow the running of proprietary tests which cannot be contributed to the central repository.

There must be a low knowledge barrier to entry for development, in order to encourage a wide variety of new developers to start contributing. In particular, we desire it to be easy to write new tests and profilers, abstracting the complexity into the core as much as possible.

We require a high level of maintainability. We want a consistent language throughout, one which is powerful and yet easy to understand when returning to the code later, not only by the author, but also by other developers.

The client must be robust, and produce consistent results. Error handling is critical—tests that do not produce reliable results are useless. Developers will never add sufficient error checking into scripts, we must have a system which fails on any error unless you take affirmative action. Where possible it should isolate hardware or harness failures from failures of the code under test; if something goes wrong in initialisation or during a test we need to know and reject that test result.

Finally, we want a consistent results architecture—it is no use to run thousands of tests if we cannot understand or parse the results. On such a scale such analysis must be fully automatable. Any results structure needs to be consistent across tests and across machines, even if the tests are being run by a wide diversity of testers.

4.2 What Tests are Needed?

As we mentioned previously, the current published automated testing is very limited in its scope. We need very broad testing coverage if we are going to catch a high proportion of problems before they reach the user population, and need those tests to be freely sharable to maximise test coverage.

Most of the current testing is performed in order to verify that the machine and OS stack is fit for a particular workload. The real workload is often difficult to set up, may require proprietary software, and is overly complex and does not give sufficiently consistent reproducible results, so use is made of a simplified simulation of that workload encapsulated within a test. This has the advantage of allowing these simulated workloads to be shared. We need tests in all of the areas below:

Build tests simply check that the kernel will build. Given the massive diversity of different architectures to build for, different configuration options to build for, and different toolchains to build with, this is an extensive problem. We need to check for warnings, as well as errors.

Static verification tests run static analysis across the code with tools like sparse, lint, and the Stanford checker, in the hope of finding bugs in the code without having to actually execute it.

Inbuilt debugging options (e.g. `CONFIG_DEBUG_PAGEALLOC`, `CONFIG_DEBUG_SLAB`) and fault insertion routines (e.g. fail every 100th memory allocation, fake a disk error occasionally) offer the opportunity to allow the kernel to test itself. These need to be a separated set of test runs from the normal functional and performance tests, though they may reuse the same tests.

Functional or unit tests are designed to exercise one specific piece of functionality. They are used to test that piece in isolation to ensure it meets some specification for its expected operation. Examples of this kind of test include LTP and Crashme.

Performance tests verify the relative performance of a particular workload on a specific system. They are used to produce comparisons between tests to either identify performance changes, or confirm none is present. Examples of these include: CPU performance with Kernbench and AIM7/reaim; disk performance with bonnie, tbench and iobench; and network performance with netperf.

Stress tests are used to identify system behaviour when pushed to the very limits of its capabilities. For example a kernel compile executed completely in parallel creates a compile process for each file. Examples of this kind of test include kernbench (configured appropriately), and deliberately running under heavy memory pressure such as running with a small physical memory.

Profiling and debugging is another key area. If we can identify a performance regression, or some types of functional regression, it is important for us to be able to gather data about what the system was doing at the time in order to diagnose it. Profilers range from statistical tools like readprofile and lockmeter to monitoring tools like vmstat and sar. Debug tools might range from dumping out small pieces of information to full blown crashdumps.

4.3 Existing Client Harnesses

There are a number of pre-existing test harnesses in use by testers in the community. Each has its features and problems, we touch on a few of them below.

IBM autobench is a fairly fully featured client harness, it is completely written in a combination of shell and perl. It has support for tests containing kernel builds and system boots. However, error handling is very complex and must be explicitly added in all cases, but does encapsulate the success or failure state of the test. The use of multiple different languages may have been very efficient for the original author, but greatly increases the maintenance overheads. Whilst it does support running multiple tests in parallel, loops within the job control file are not supported nor is any complex "programming."

OSDL STP The Open Systems Development Lab (OSDL) has the Scalable Test Platform (STP). This is a fully integrated testing environment with both a server harness and client wrapper. The client wrapper here is very simple consisting of a number of shell support functions. Support for reboot is minimal and kernel installation is not part of the client. There is no inbuilt handling of the meaning of results. Error checking is down to the test writer; as this is shell it needs to be explicit else no checking is performed. It can operate in isolation and results are emailable, reboot is currently being added.

LTP[1] The Linux Test Project is a functional / regression test suite. It contains approximately 2900 small regression tests which are applied to the system running LTP. There is no support for building kernels or booting them, performance testing or profiling. Whilst it contains a lot of useful tests, it is not a general heavy weight testing client.

A number of other testing environments currently exist, most appear to suffer from the same basic issues, they evolved from the simplest possible interface (a script) into a test suite; they were not designed to meet the level of requirements we have identified and specified.

All of those we have reviewed seem to have a number of key failings. Firstly, most lack most lack bottom up error handling. Where support exists it must be handled explicitly, testers never will think of everything. Secondly, most lack consistent machine parsable results. There is often no consistent way to tell if a test passes, let alone get any details from it. Lastly, due to their evolved nature they are not easy to understand nor to maintain. Fortunately it should be reasonably easy to wrap tests such as LTP, or to port tests from STP and autobench.

4.4 Autotest a Powerful Open Client

The Autotest open client is an attempt to address the issues we have identified. The aim is to produce a client which is open source, implicitly handles errors, produces consistent results, is easily installable, simple to maintain and runs either standalone or within any server harness.

Autotest is an all new client harness implementation. It is completely written in Python; chosen for a number of reasons, it has a simple, clean and consistent syntax, it is object oriented from inception, and it has very powerful error and exception handling. Whist no language is perfect, it meets the key design goals well, and it is open source and widely supported.

[1] `http://ltp.sourceforge.net/`

As we have already indicated, there are a number of existing client harnesses; some are even open-source and therefore a possible basis for a new client. Starting from scratch is a bold step, but we believe that the benefits from a designed approach outweigh the effort required initially to get to a workable position. Moreover, much of the existing collection of tests can easily be imported or wrapped.

Another key goal is the portability of the tests and the results; we want to be able to run tests anywhere and to contribute those test results back. The use of a common programming language, one with a strict syntax and semantics should make the harness and its contained tests very portable. Good design of the harness and results specifications should help to maintain portable results.

4.5 The autotest Test Harness

Autotest utilises an executable control file to represent and drives the users job. This control file is an executable fragment of Python and may contain any valid Python constructs, allowing the simple representation of loops and conditionals. Surrounding this control file is the Autotest harness, which is a set of support functions and classes to simplify execution of tests and allow control over the job.

The key component is the job object which represents the executing job, provides access to the test environment, and provides the framework to the job. It is responsible for the creation of the results directory, for ensuring the job output is recorded, and for any interactions with any server harness. Below is a trivial example of a control file:

```
job.runtest('test1', 'kernbench', 2, 5)
```

One key benefit of the use of a real programming language is the ability to use the full range of its control structures in the example below we use an iterator:

```
for i in range(0, 5):
    job.runtest('test%d' % i, 'kernbench',
        2, 5)
```

Obviously as we are interested in testing Linux, support for building, installing and booting kernels is key. When using this feature, we need a little added complexity to cope with the interruption to control flow caused by the system reboot. This is handled using a phase stepper which maintains flow across execution interruptions, below is an example of such a job, combining booting with iteration:

```
def step_init():
    step_test(1)

def step_test(iteration):
    if (iteration < 5):
        job.next_step([step_test,
                      iteration + 1])

    print "boot: %d" % iteration

    kernel = job.distro_kernel()
    kernel.boot()
```

Tests are represented by the test object; each test added to Autotest will be a subclass of this. This allows all tests to share behaviour, such as creating a consistent location and layout for the results, and recording the result of the test in a computer readable form. In figure 4 is the class definition for the kernbench benchmark. As we can see it is a subclass of test, and as such benefits from its management of the results directory hierarchy.

4.6 Summary

We feel that Autotest is much more powerful and robust design than the other client harnesses available, and will produce more consistent results. Adding tests and profilers is simple, with a low barrier to entry, and they are easy to understand and maintain.

Much of the power and flexibility of Autotest stems from the decision to have a user-defined control file, and for that file to be written in a powerful scripting language. Whilst this was more difficult to implement, the interface the user sees is still simple. If the user wishes to repeat tests, run tests in parallel for stress, or even write a bisection search for a problem inside the control file, that is easy to do.

The Autotest client can be used either as standalone, or easily linked into any scheduling backend, from a simple queueing system to a huge corporate scheduling and allocation engine. This allows us to leverage the resources of larger players, and yet easily allow individual developers to reproduce and debug problems that were found in the lab of a large corporation.

Each test is a self-contained modular package. Users are strongly encouraged to create open-source tests (or wrap existing tests) and contribute those to the main test repository on `test.kernel.org`.[2] However, private tests and repositories are also allowed, for maximum flexibility. The modularity of the tests means that different maintainers can own and maintain each test, separate from the core harness. We feel this is critical to the flexibility and scalability of the project.

We currently plan to support the Autotest client across the range of architectures and across the

[2]See the autotest wiki `http://test.kernel.org/autotest`.

```
import test
from autotest_utils import *

class kernbench(test):

    def setup(self,
              iterations = 1,
              threads = 2 * count_cpus(),
              kernelver = '/usr/local/src/linux-2.6.14.tar.bz2',
              config =  os.environ['AUTODIRBIN'] + "/tests/kernbench/config"):

        print "kernbench -j %d -i %d -c %s -k %s" % (threads, iterations, config, kernelver)

        self.iterations = iterations
        self.threads = threads
        self.kernelver = kernelver
        self.config = config

        top_dir = job.tmpdir+'/kernbench'
        kernel = job.kernel(top_dir, kernelver)
        kernel.config([config])


    def execute(self):
        testkernel.build_timed(threads)         # warmup run
        for i in range(1, iterations+1):
            testkernel.build_timed(threads, '../log/time.%d' % i)

        os.chdir(top_dir + '/log')
        system("grep elapsed time.* > time")
```

Figure 4: Example test: kernbench

main distros. There is no plans to support other operating systems, as it would add unnecessary complexity to the project. The Autotest project is released under the GNU Public License.

5 Future

We need a broader spectrum of tests added to the Autotest project. Whilst the initial goal is to replace autobench for the published data on `test.kernel.org`, this is only a first step—there are a much wider range of tests that could and should be run. There is a wide body of tests already available that could be wrapped and corralled under the Autotest client.

We need to encourage multiple different entities to contribute and share testing data for maximum effect. This has been stalled waiting on the Autotest project, which is now nearing release, so that we can have a consistent data format to share and analyse. There will be problems to tackle with quality and consistency of data that comes from a wide range of sources.

Better analysis of the test results is needed. Whilst the simple red/yellow/green grid on `test.kernel.org` and simple gnuplot graphs are surprisingly effective for so little effort, much more could be done. As we run more tests, it will become increasingly important to summarise and fold the data in different ways in order to make it digestible and useful.

Testing cannot be an island unto itself—not only must we identify problems, we must communicate those problems effectively and efficiently back to the development community, provide them with more information upon request, and be able to help test attempted fixes.

We must also track issues identified to closure.

There is great potential to automate beyond just identifying a problem. An intelligent automation system should be able to further narrow down the problem to an individual patch (by bisection search, for example, which is O(log2) number of patches). It could drill down into a problem by running more detailed sets of performance tests, or repeating a failed test several times to see if a failure was intermittent or consistent. Tests could be selected automatically based on the area of code the patch touches, correlated with known code coverage data for particular tests.

6 Summary

We are both kernel developers, who started the both `test.kernel.org` and Autotest projects out of a frustration with the current tools available for testing, and for fully automated testing in particular. We are now seeing a wider range of individuals and corporations showing interest in both the `test.kernel.org` and Autotest projects, and have high hopes for their future.

In short we need:

- more automated testing, run at frequent intervals,
- those results need to be published consistently and cohesively,
- to analyse the results carefully,
- better tests, and to share them, and
- a powerful, open source, test harness that is easy to add tests to.

There are several important areas where interested people can help contribute to the project:

- run a diversity of tests across a broad range of hardware,
- contribute those results back to `test.kernel.org`,
- write new tests and profilers, contribute those back, and
- for the kernel developers ... fix the bugs!!!

An intelligent system can not only improve code quality, but also free developers to do more creative work.

Acknowledgements

We would like to thank OSU for the donation of the server and disk space which supports the `test.kernel.org` site.

We would like to thank Mel Gorman for his input to and review of drafts of this paper.

Legal Statement

This work represents the view of the authors and does not necessarily represent the views of either Google or IBM.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be the trademarks or service marks of others.

Linux Laptop Battery Life

Measurement Tools, Techniques, and Results

Len Brown Konstantin A. Karasyov

Vladimir P. Lebedev Alexey Y. Starikovskiy

Intel Open Source Technology Center

`{len.brown,konstantin.a.karasyov}@intel.com`
`{vladimir.lebedev,alexey.y.starikovskiy}@intel.com`

Randy P. Stanley
Intel Mobile Platforms Group
`randy.p.stanley@intel.com`

Abstract

Battery life is a valuable metric for improving Linux laptop power management.

Battery life measurements require repeatable workloads. While BAPCo® MobileMark® 2005 is widely used in the industry, it runs only on Windows® XP. Intel's Open Source Technology Center has developed a battery life measurement tool-kit for making reliable battery life measurements on Linux® with no special lab equipment necessary.

This paper describes this Linux battery life measurement tool-kit, and some of the techniques for measuring Linux laptop power consumption and battery life.

This paper also includes example measurement results showing how selected system configurations differ.

1 Introduction

First we examine common industry practice for measuring and reporting laptop battery life.

Next we examine the methods available on ACPI-enabled Linux systems to measure the battery capacity and battery life.

Then we describe the implementation of a battery life measurement toolkit for Linux.

Finally, we present example measurement results applying this toolkit to high-volume hardware, and suggest some areas for further work.

1.1 State of the Industry

Laptop vendors routinely quote MobileMark® battery measurement results when introducing new systems. The authors believe that this is not only the most widely employed industry measurement, but that MobileMark also reflects best known industry practice. So we will focus this section on MobileMark and ignore what we consider lesser measurement programs.

1.2 Evolution of MobileMark®

In 1995, BAPCo®, the Business Applications Performance Corporation, introduced a battery-life (BL) workload to support application based power evaluation. The first incarnation, SYSmark BL, was a Windows® 3.1 based workload which utilized office applications to implement a repeatable workload to produce a "battery run down" time. Contrary to performance benchmarks which executed a stream of commands, this workload included delays which were intended to represents real user interaction, much like a player piano represent real tempos. Because the system is required to deplete its own battery, a master system and physical interface was required as well as the slave system under test. In late 1996 the workload was re-written to support Windows95 adapted to 32-bit applications.

When Windows® 98 introduced ACPI support, BAPCo overhauled the workload to shed the cumbersome and expensive hardware interface. SYSmark98 BL became the first software only BL workload. (No small feat as the system was now required to resurrect itself and report BL without adding additional overhead.) Additionally a more advanced user delay model was introduced and an attempt was made to understand the power performance trade-off within mobile systems by citing the number of loops completed during the life of the battery. Although well intended, this qualification provided only gross level insight into the power performance balance of mobile systems.

In 2002, BAPCo released MobileMark 2002 [MM02] which modernized the workload and adopted a response based performance qualifier which provided considerably more insight into the power performance balance attained by modern power management schemes. Additionally they attempted to better define a more level playing field by providing a more rigorous set of system setting requirements and recommendations, and strongly recommending a light meter to calibrate the LCD panel brightness setting to a common value. Additionally, they introduced a "Reader" module to complement the Office productivity module. Reader provided a time metric for an optimal BL usage model to define a realistic upper bound while executing a real and practical use.

In 2005 BAPCo's MobileMark 2005 [MM05] added to the MobileMark 2002 BL "suite" by introducing new DVD and Wireless browsing modules as well as making slight changes to increase robustness and hold the work/time constant for all machines. Today these modules help us to better understand the system balance of power and performance. Multiple results also form contour of solutions reflective of the respective user and usage models.

1.3 Learning from MobileMark® 2005

While MobileMark is not available for Linux, it illustrates some of the best industry practices for real use power analysis that Linux measurements should also employ.

1.3.1 Multiple Workloads

Mobile systems are subject to different user[1] and usage models,[2] each with its own battery

[1]Different users type, think and operate the system differently.

[2]Usage models refers to application choices and content.

life considerations. To independently measure different usage models, MobileMark 2005 provides 4 workloads:

1. Office productivity 2002SE

   This workload is the second edition of MobileMark 2002 Office productivity. Various office productivity tools are used to open and modify office documents.

   Think time is injected between various operations to reflect that real users need to look at the screen and react before issuing additional input.

   The response time of selected operations is recorded (not including delays) to be able to qualify the battery life results and differentiate the performance level available while attaining that battery life.

2. Reader 2002SE

   This workload is a second edition of MobileMark 2002 Reader. Here, a web browser reads a book from local files, opening a new page every 2 minutes. This workload is almost completely idle time, and can be considered an upper bound, which no "realistic" activity can possibly exceed.

3. DVD Playback 2005

   InterVideo® WinDVD® plays a reference DVD movie repeatedly until the battery dies. WinDVD monitors that the standard frame rate, so that the harness can abort the test if the work level is not sustained. In practice, modern machines have ample capacity to play DVDs, and frames are rarely dropped.

4. Wireless browsing 2005

   Here the system under test loads a web page every 15 seconds until the battery dies. The web pages are an average of 150 KB. This workload is not specific to wireless networks, however, and in theory could be run over wired connections.

1.3.2 Condition the Battery

In line with manufacturer's recommendations, BAPCo documentation recommends conditioning the battery before measurement. This entails simply running the battery from full charge until full discharge at least once.

For popular laptop batteries today, conditioning tends to minimize memory effects, extend the battery life, and increase the consistency of measurements.

MobileMark recommends conditioning the battery before taking measurements.

1.3.3 Run the Battery until fully Discharged

Although conditioning tends to improve the accuracy of the internal battery capacity instrumentation, this information is not universally accurate or reliable before or after conditioning.

MobileMark does not trust the battery instrumentation, and disables the battery low-capacity warnings. It measures battery life by running on battery power until the battery is fully discharged and the system crashes.

1.3.4 Qualify Battery Life with Performance

In addition to the battery life (in minutes) MobileMark Office productivity results always report response time.

This makes it easy to tell the difference between a battery life result for a low performance system and a similar result for a high performance system that employs superior power management.

There is no performance component reported for the other workloads, however, as the user experience for those workloads is relatively insensitive to performance.

1.3.5 Constant Work/Time

The MobileMark Office productivity workload was calibrated to a minimal machine that completed one workload iteration in about 90 minutes. If a faster machine completes the workload iteration in less time, the system idles until the next activity cycle starts at 90-minutes.

2 Measurement Methods

Here we take a closer look at the methods available to observe and measure battery life in a Linux context.

2.1 Using an AC Watt Meter

Consumer-grade Watt Meters with a resolution of 0.1Watt and 1-second sampling rate are available for about 100 U.S. Dollars.[3] While intended to tell you the cost of operating your old refrigerator, they can just as easily tell you the A/C draw for a computer.

It is important to avoid the load of battery charging from this scenario by measuring. This can be done by measuring only when the battery is fully charged, or for laptops that allow it, running on A/C with the battery physically removed.

You'll be able to see the difference between such steady-state operations as LCD on vs. off, LCD brightness, C-states and P-states. However, it will be very difficult to observe transient behavior with the low sampling rate.

Unfortunately, the A/C power will include the loss in the AC-to-DC power supply "brick." While an "External Power Adapter" sporting an Energy Star logo[4] rated at 20 Watts or greater will be more than 75% efficient, others will not meet that criteria and that can significantly distort your measurement results.

So while this method is useful for some types of comparisons, it isn't ideal for predicting battery life. This is because most laptops behave differently when running on DC battery vs. running on AC. For example, it is extremely common for laptops to enable deep C-states only on DC power and to disable them on AC power.

2.2 Using a DC Watt Meter on the DC converter

It is possible to modify the power adapter by inserting a precise high-wattage low-ohm resistor in series on the DC rail and measuring the voltage drop over this resistor to calculate the current, and thus Watts.

This measurement is on the DC side of the converter, and thus avoids the inaccuracy from AC-DC conversion above. But this method suffers the same basic flaw as the AC meter method above, the laptop is running in AC mode, and that is simply different from DC mode.

[3]Watt's Up Pro: https://www.doubleed.com

[4]http://www.energystar.gov

2.3 Replacing the Battery with a DC power supply

The next most interesting method to measure battery consumption on a laptop is to pull apart the battery and connect it to a lab-bench DC power supply.

This addresses the issue of the laptop running in DC mode. However, few reading this paper will have the means to set up this supply, or the willingness to destroy their laptop battery.

However, for those with access this type of test setup, including a high-speed data logger; DC consumption rates can be had in real-time, with never a wait for battery charging.

Further, it is possible that system designers may choose to make the system run differently depending on battery capacity. For example, high-power P-states may be disabled when on low battery power—but these enhancements would be disabled when running on a DC power supply that emulates a fully charged battery.

2.4 Using a DC Watt Meter on an instrumented battery

Finally, it is possible to instrument the output of the battery itself. Like the DC power supply method above, this avoids the issues with the AC wattmeter and the instrumented power converter method in that the system is really running on DC. Further, this allows the system to adapt as the battery drains, just as it would in real use. But again, most people who want to measure power have neither a data logger, nor a soldering iron available.

2.5 Using Built-in Battery Instrumentation

Almost all laptops come with built in battery instrumentation where the OS read capacity, calculate drain and charge rates, and receive capacity alarms.

On Linux, `/proc/acpi/battery/*/info` and `state` will tell you about your battery and its current state, including drain rate.

Sometimes the battery drain data will give a good idea of average power consumption, but often times this data is mis-leading.

One way to find out if your drain rate is accurate is to plot the battery capacity from fully charged until depleted. If the system is running a constant workload, such as idle, then the instrumentation should report full capacity equal to the design capacity of the battery at the start, and it should report 0 capacity just as the lights go out—and it should report a straight line in between. In practice, only new properly conditioned batteries do this. Old batteries and batteries that have not been conditioned tend to supply very poor capacity data.

Figure 1 shows a system with an old $(4.4AH * 10.4V) = 47.520\,Wh$ battery. After fully charging the battery, the instrumentation at the start of the 1st run indicates that the battery capacity of under 27.000 Wh. If the battery threshold warning was enabled for that run, the system would have shut down well before 5,000 seconds—even though the battery actually lasted past 7,000 seconds.

The 1st run was effectively conditioning the battery. The 2nd run reported a fully charged capacity of nearly 33.000 Wh. The actual battery life was only slightly longer than the initial conditioning run, but in this case the reported capacity was closer to the truth. The

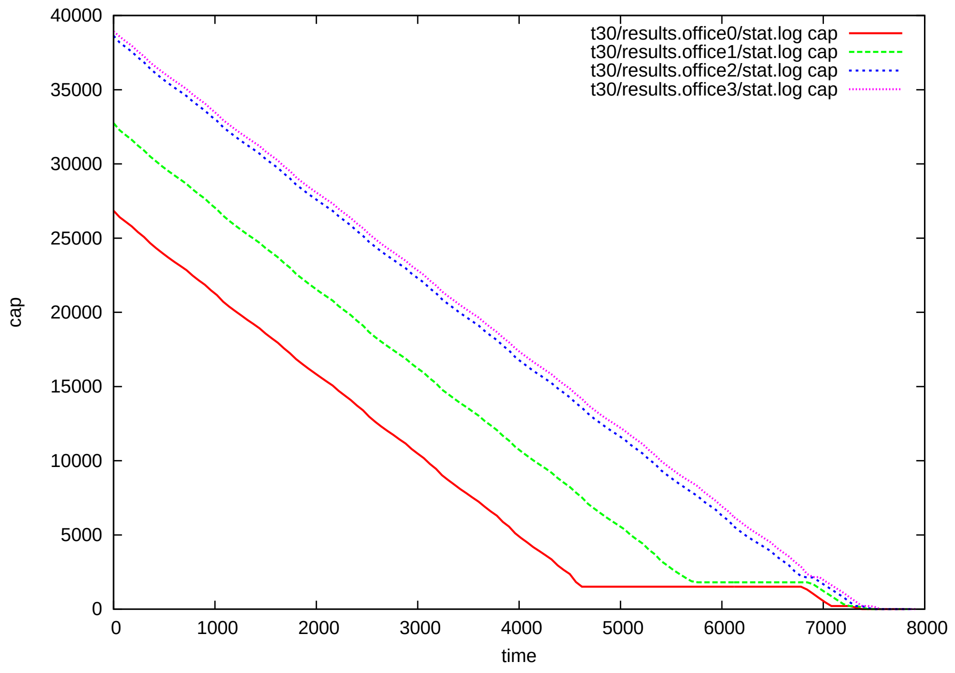


Figure 1: System A under-reports capacity until conditioned

3rd run started above 38.000 Wh and was linear from there until the battery died after 7,000 seconds. The 4th run showed only marginally more truthful results. Note that a 10% battery warning at 4,752 would actually be useful to a real user after the battery has been conditioned.

Note also that the slope of all 4 lines is the same. In this case, the rate of discharge shown by the instrumentation appears accurate, even for the initial run.

The battery life may not be longer than the slope suggests, it may be shorter. Figure2 shows system B suddenly losing power near the end of its conditioning run. However, the 2nd (and subsequent) runs were quite well behaved.

Figure 3 shows system C with a drop-off that is sure to fool the user's low battery trip points. In this case the initial reported capacity does not change, staying at about 6800 of 71.000 Wh (95%). However, the first run drops off a cliff at about 11,000 seconds. The second and third runs drop at about 13,500. But subsequent runs all drop at about 12,000 seconds. So conditioning the battery didn't make this one behave any better.

Finally, Figure 4 shows system D reporting initial capacity equal to 100% of its 47.950 Wh design capacity. But upon use, this capacity drops almost immediately to about 37.500 Wh. Even after being conditioned 5 times, the battery followed the same pattern. So either the initial capacity was correct and the drain rate is wrong, or initial capacity is incorrect and the drain rate is correct. Note that this behavior went away when a new battery was used. A new battery reported 100% initial capacity, and 0% final capacity, connected by a straight line.

In summary, the only reliable battery life measurement is a wall clock measurement from full charge until the battery is depleted. Depleted here means ignoring any capacity warnings and running until the lights go out.

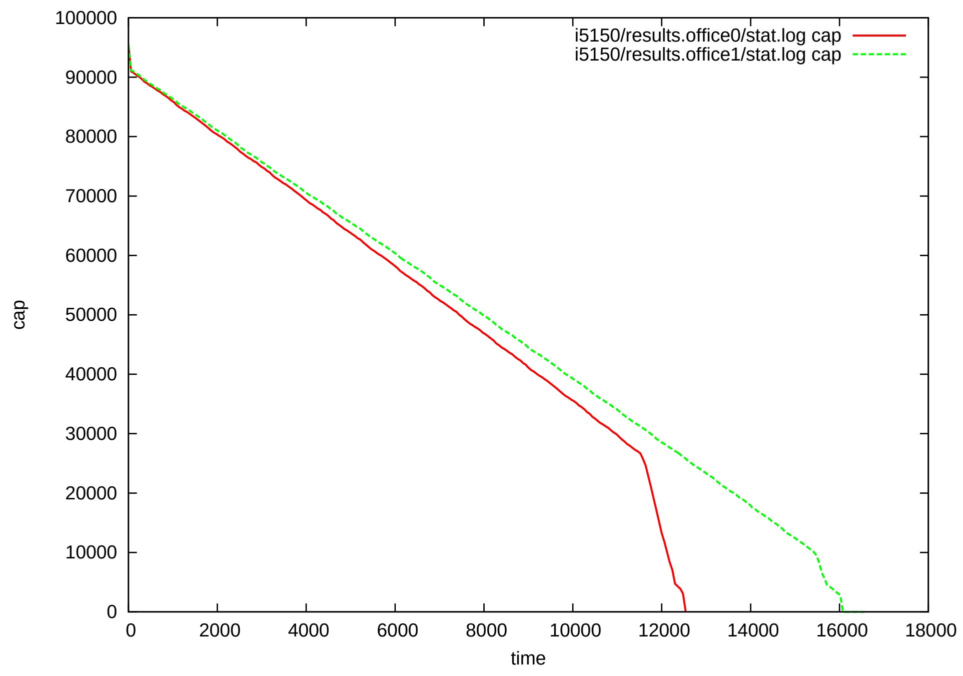


Figure 2: System B over-reports capacity until conditioned

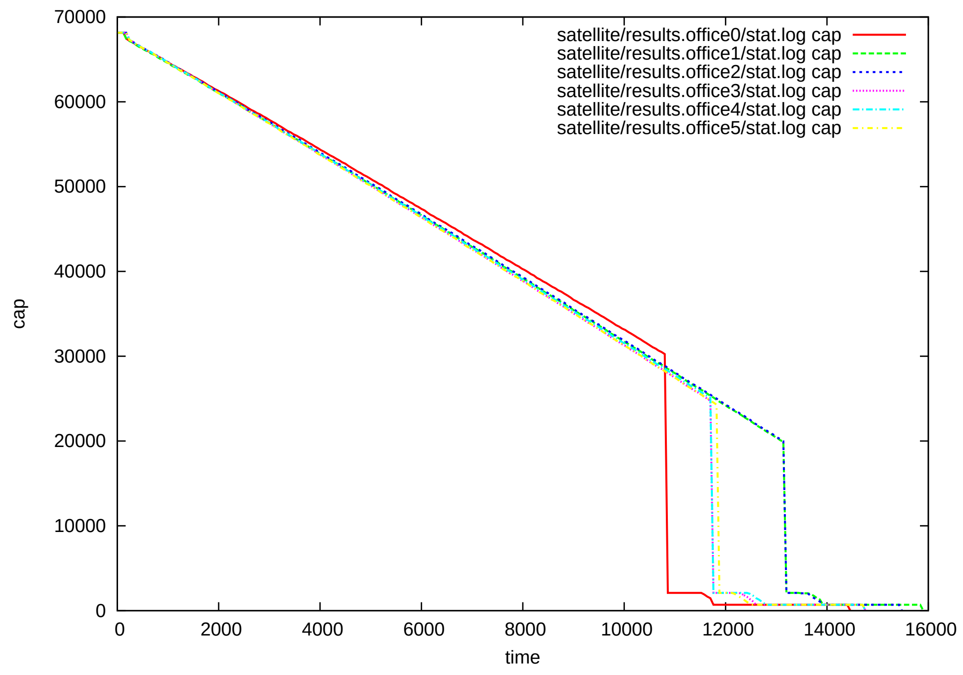


Figure 3: System C over-reports final capacity, conditioning does not help

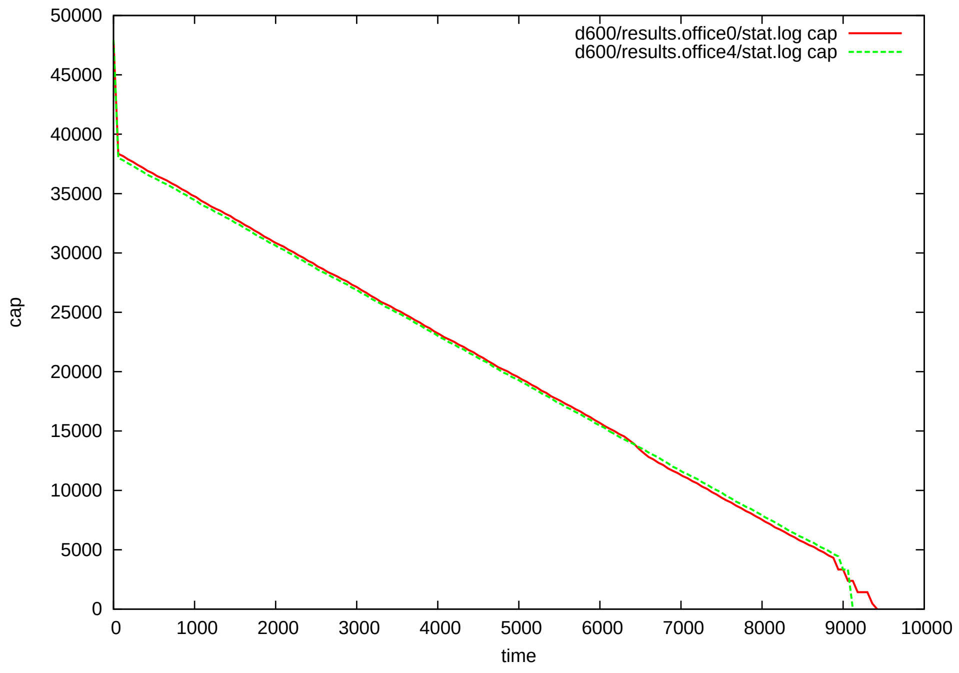


Figure 4: System D over-reports initial capacity, conditioning does not help

3 Linux Battery Life Toolkit

The Linux Battery Life Toolkit (bltk) consists of a test framework and six example workloads. There are common test techniques that should be followed to assure repeatable results no matter what the workload.

3.1 Toolkit Framework

The toolkit framework is responsible for launching the workload, collecting statistics during the run, and summarizing the results after a test completes.

The framework can launch any arbitrary workload, but currently has knowledge of 6 example workloads: Idle, Reader, Office, DVD Player, SW Developer, and 3D-Gamer.

3.2 Idle Workload

The idle workload simply executes the framework without invoking any programs. Statistics are collected the same way as for the other workloads.

3.3 Web Reader Workload

The web reader workload opens an HTML-formatted version of *War and Peace* by Leo Tolstoy[5] in Firefox® and then sends "next page" keyboard events to browser every two minutes, simulating interaction with the human reader.

[5]We followed the lead of BAPCo's MobileMark here on selection of reading material.

3.4 Open Office Workload

Open Office rev 1.1.4 was chosen for this toolkit because it is stable and freely available. It is intended to be automatically installed by the toolkit to avoid results corruption due to local settings and version differences.

3.4.1 Open Office Activities

Currently 3 applications from OpenOffice suite are used for the Office workload, `oowriter`, `oocalc` and `oodraw`. The set of common operations is applied to these applications to simulate activities, typical for office application users.

Using `oowriter`, the following operations are performed:

- text typing
- text pattern replacement
- file saving

Using `oocalc`, the following operations are performed:

- creating spreadsheet;
- editing cells values;
- assigning math expression to the cell;
- expanding math expression over a set of cells;
- assigning set of cells to the math expression;
- file saving;

Using `oodraw`, the following operations are performed:

- duplicating image;
- moving image over the document;
- typing text over the image;
- inserting spreadsheet;
- file saving.

3.4.2 Open Office User Input

User input consists of actions and delays. Actions are represented by the key strokes sent to the application window though the X server. This approach makes the application perform the same routines as it does during interaction with the real user.[6] Delays are inserted between actions to represent a real user.

The Office workload scenario is not hard-coded, but is scripted using the capabilities shown in Appendix A.

3.4.3 Open Office Performance Scores

A single iteration of the office workload scenario completes in 720 seconds. When a faster machine completes the workload in less than 720 seconds, it is idle until the next iteration starts.

$$720 seconds = Workload\_time + Idle$$

Workload time consists of Active_time—the time it takes for the system to start applications and respond to user commands—plus the delays that the workload inserts to model user type-time and think-time.

[6]The physical input device, such as keyboard and mouse are not used here.

$$Workload\_time = Active\_time + Delay\_time$$

So the performance metric for each workload scenario iteration is Active_time, which is calculated by measuring Workload_time and simply and subtracting the known Delay_time.

The reported performance score is the average Active_time over all loop iterations, normalized to a reference Active_time so that bigger numbers represent better performance:

$$Performance\_score = 100 * Active\_reference / Average\_Active\_measured$$

3.5 DVD Movie Playback Workload

`mplayer` is invoked to play a DVD movie until the battery dies. Note that `mplayer` does not report frame rate to the toolkit framework. For battery life comparisons, equal work/time must be maintained, so that it is assumed, but not verified in these tools that modern systems can play DVD movies at equal frame rates.

3.6 Software Developer Workload

The software developer workload mimics a Linux ACPI kernel developer: it invokes `vi` to insert a comment string into one of the Linux ACPI header files and then invokes `make -j N` on a Linux kernel source tree, where N is chosen to be three times the number of processors in a system. This cycle is extended out to 12 minutes with idle time to more closely model constant work/time on different systems.

The Active_time for the developer workload is the time required for the `make` command to complete, and it is normalized into a performance score the same was as for the Office workload.

$$Performance\_score = 100 * Active\_reference / Average\_Active\_measured.$$

3.7 3D Gamer Workload

A 3D gamer workload puts an entirely different workload on a laptop, one that is generally more power-hungry than all the workloads above.

However, we found that 3D video support is not universally deployed or enabled in Linux, nor are there a lot of selections of games that are simultaneously freely available, run on a broad range of platforms, and include a demo-mode that outputs performance.

`glxgears` satisfies the criteria for being freely available, universally supported, and it reports performance; however, that performance is not likely to closely correlate to what a real 3D game would see. So we are not satisfied that we have a satisfactory 3D-Gamer metric yet.

In the case of a 3D game workload, a reasonable performance metric to qualify battery life would be based on frames/second.

$$3D\_Performance\_Score = FPS\_measured / FPS\_reference$$

4 Example Measurement Results

This section includes example battery life measurements to show what a typical user can do on their system without the aid of any special instrumentation.

Unless otherwise specified, the Dell Inspiron™ 6400 shown in Table 1 was used as the example system.

Note that this system is a somewhat arbitrary reference. It has a larger and brighter screen

System	Dell Inspiron 6400
Battery	53 Wh
Processor	Intel Core Duo T2500, 2GHz 2MB cache, 667MHz bus
LCD	15.4" WXGA, min bright
Memory	1GB DDR2, 2DIMM, 533MHz
Distribution	Novell SuSE 10.1 BETA
GUI	KDE
Linux	2.6.16 or later
HZ	250
cpufreq	ondemand governor
Battery Alerts	ignored
Screen Saver	disabled
DPMS	disabled
Wired net	disabled
Wireless	disabled

Table 1: Nominal System Under Test

than many available on the market. It arrived with a 53 Wh 6-cell battery, but is also available with an 85 Wh 9-cell battery, which would increase the absolute battery life results by over 50%. But the comparisons here are generally of this system to itself, so these system-specific parameters are equal on both sides.

4.1 Idle Workload

4.1.1 Idle: Linux vs. Windows

Comparing Linux[7] with Windows[8] on the same hardware tells us how Linux measures up to high-volume expectations.

This baseline comparison is done with pure-idle workload. While trivial, this "workload" is also crucial, because a difference in idle power consumption will have an effect on virtually all other workloads.

[7]Linux-2.6.16+ as delivered with Novell SuSE 10.1 BETA

[8]Windows® XP SP2

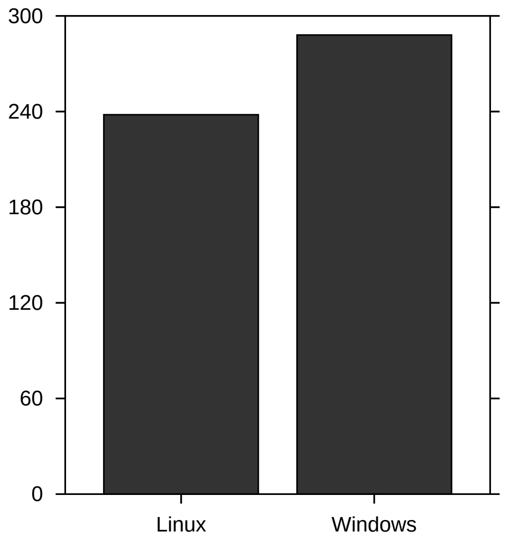


Figure 5: Idle: Linux vs. Windows

Here the i6400 lasts 288 minutes on Windows, but only 238 minutes on Linux, a 50 minute deficit. One can view this as a percentage, eg. Linux has $238/288 = 83\%$ of the idle battery life as compared to Windows.

One can also estimate the average power using the fixed 53 Wh battery capacity. $(53Wh * 60min/Hr)/288min = 11.0W$ for Windows. $(53Wh * 60min/Hr)/238min = 13.4W$ for Linux. So here Linux is at a 2.4W deficit compared to Windows in idle.

4.1.2 Idle: The real cost of the LCD

While the i6400 has a larger than average display, the importance of LCD power can not be over-stated—even for systems with smaller displays.

The traditional screen saver that draws pretty pictures on an idle screen is exactly the opposite of what you want for long battery life. The reason isn't because the display takes more power, the reason is because it takes the proces-

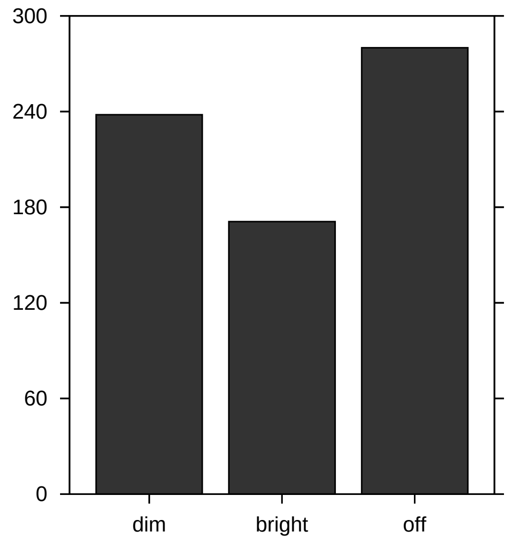


Figure 6: Idle: Effect of LCD

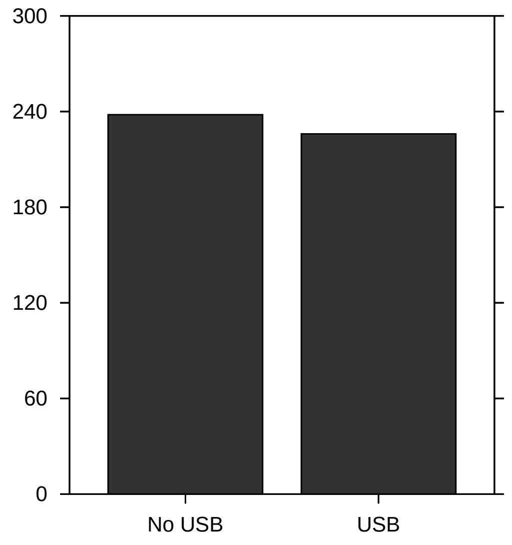


Figure 7: Idle: Effect of USB

sor out of the deepest available C-state when there is nothing "useful" to do.

The CPU can be removed from that equation by switching to a screen saver that does not run any programs. Many select the "blank" screen saver on the assumption that it saves power—but it does not. A Black LCD actually saves no power vs. a white LCD. This is because the black LCD has the backlight on just as bright as the white LCD, but it is actually using fractionally more energy to block that light with every pixel.

So the way to save LCD power is to dim the back-light so it is no brighter than necessary to read the screen; and or turn it off completely when you are not reading at the screen. Note that an LCD that is *off* should appear *black* in a darkened room. If it is glowing, then the pixels are simply fighting to obscure a backlight that is still on. A screen saver that runs no programs and has DPMS (Display Power Management Signaling) enabled to turn off the display is hugely important to battery life.

On the example system, the 238-minute "dim" idle time drops to 171 for maximum LCD brightness, and increases to 280 minutes for LCD off. Expressed as Watts, dim is 13.4W, bright is 18.6W, and off is 11.4W. So this particular LCD consumes between 2.0 and 7.2W. Your mileage will vary.

Note that because of its large demands system power, analysis of the power consumption of the other system components is generally most practical when the LCD is off.

4.1.3 Idle: The real cost of USB

The i6400 has no integrated USB devices. So if you execute `lsusb`, you'll see nothing until you plug in an external device.

If an (unused) USB 1.0 mouse is connected to the system, battery life drops 12 minutes to 226 from 238. This corresponds to $(14.1 - 13.4) = 0.7W$.

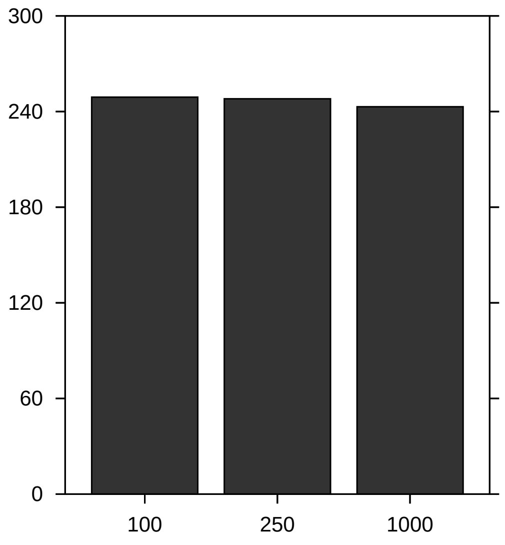


Figure 8: Idle: Effect of HZ

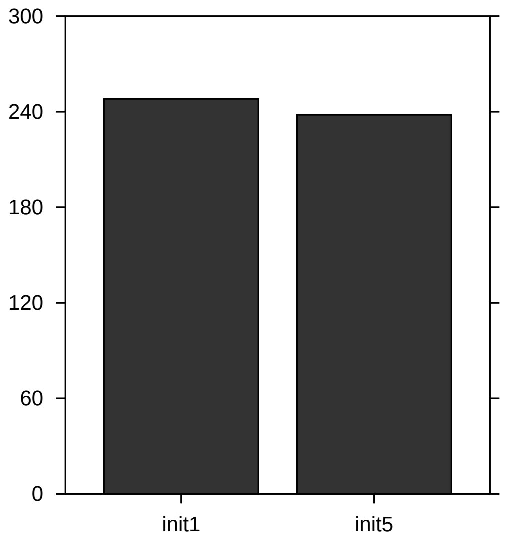


Figure 9: Idle: init1 vs. init5

4.1.4 Idle: Selecting HZ

In Linux-2.4, the periodic system timer tick ran at 100 HZ. Linux-2.6 started life running at 1000 HZ. Linux-2.6.13 added `CONFIG_HZ`, with selections of 100, 1000, and a compromise default of 250 HZ.

Figure 8 shows that the selection of HZ has a very small effect on the i6400, though others have reported larger differences on other systems. Note that since this difference was small, this comparison was made in single-user mode.

4.1.5 Idle: init1 vs. init5

The Linux vs. Windows measurement above was in multi-user GUI mode—though the network was disabled. One question often asked if the GUI (KDE, in this example) and other standard daemons have a significant effect on Linux battery life.

In this example, the answer is yes, but not much. Multi-user battery life is 238 minutes, and Single-user battery life is 10 minutes longer at 248—only a 4% difference. Expressed as Watts, $13.4 - 12.8 = 0.6W$ to run in multi-user GUI mode.

However, init5 battery consumption may depend greatly on how the administrator configures the system.

4.1.6 Idle: 7200 vs. 5400 RPM Disk Drives

The i6400 arrived with a 5400 RPM 40GB Fujitsu MHT2040BH SATA drive. Upgrading that drive to a 7200 RPM 60GB Hitachi HTS721060G9SA00 SATA drive reduced single-user[9] idle battery life by 16 minutes, to 232 from 248 (6%). This corresponds to an average power difference of 0.89W. The specifications for the drives show the Hitachi consuming about 0.1W more in idle and standby, and the same for read/write. So it is not im-

[9]init1 idle is used as the baseline here because the difference being measured is small, and to minimize the risk that the two different drives are configured differently.

mediately clear why Linux loses an additional 0.79W here.

4.1.7 Idle: Single Core vs. Idle Dual Core

Disabling one of the cores by booting with `maxcpus=1` has no measurable effect on idle battery life. This is because the BIOS leaves the cores in the deepest available C-state. When Linux neglects to start the second core, it behaves almost exactly as if Linux had started that core and entered the deepest available C-state on it.

Note that taking a processor off-line at run-time in Linux does not currently put that processor into the deepest available C-state. There is a bug[10] where offline processors instead enter C1. So taking a processor offline at run-time can actually result in worse battery life than if you leave it alone and let Linux go idle automatically.

4.1.8 The case against Processor Throttling (T-States)

Processor Throttling States (T-states) are available to the administrator under `/proc/acpi/processor/*/throttling` to modulate the clock supplied to the processors.

Most systems support 8 throttling states to decrease the processor frequency in steps of 12.5%. Throttling the processor frequency is independent of P-state frequency changes, so the two are combined. For the example, Table 2 shows the potential effect of throttling when the example system is in P0 or P3.

[10] `http://bugzilla.kernel.org/show_bug.cgi?id=5471`

State	P0 MHz	P3 MHz
T0	2000	1000
T1	1750	875
T2	1500	750
T3	1250	625
T4	1000	500
T5	750	375
T6	500	250
T7	250	125

Table 2: Throttling States for the Intel® Core™ Duo T2500

Throttling has an effect on processor frequency only when the system is in the C0 state executing instructions. In the idle loop, Linux is in the Cx state (*x: x != 0*) where no instructions are executed and throttling has no effect, as the clock is already stopped.

Indeed, T-states have been shown to have a net *negative* impact on battery life on some systems, as they can interfere with the mechanisms to efficiently enter deep C-states.

On the example system, throttling the idle system down to the T7, the slowest speed, had a net *negative* impact on idle battery life of 4 minutes.

Throttling is used by Linux for passive cooling mode and for thermal emergencies. It is not intended for the administrator to use throttling to maximize performance/power or extend battery life. That is what cpufreq processor performance states are for. So the next time you are exploring the configuration menus of the powersavd GUI, do NOT check the box that enables processor clock throttling. It is a bug that the administrator is given the opportunity to make that mistake.

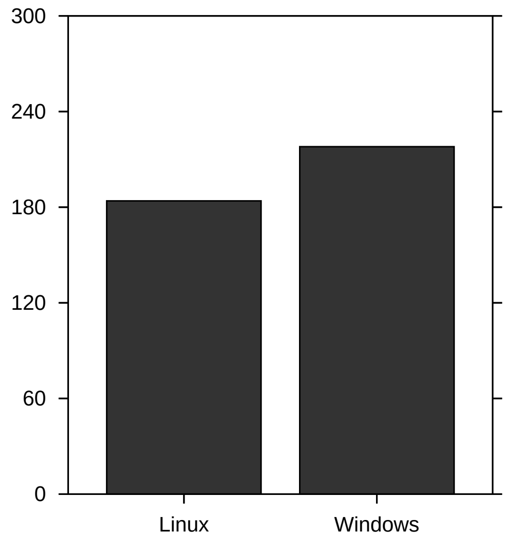


Figure 10: DVD: Linux vs. Windows

4.2 Reader Workload equals Init 5 Idle

Adding the Reader workload to init5 idle results in exactly the same battery life—238 minutes. The bar chart is left as an exercise for the reader.

4.3 DVD Movie Workload

4.3.1 DVD Movie on Linux vs. Windows

The DVD movie playback workload is also attractive for comparing Linux and Windows. This constant work/time workload leaves little room for disagreement about what the operating environment is supplying to the user. DVD movie playback is also a realistic workload, people really do sit down and watch DVDs on battery power.

However, different DVD player software is used in each operating environment. The Windows solution uses WinDVD®, and the Linux measurement uses mplayer.

Here the i6400 plays a DVD on Linux for 184 minutes (3h4m). The i6400 plays the same DVD on Windows for 218 minutes (3h38m). This 34 minute deficit puts Linux at about 84% of Windows. In terms of Watts, Linux is at a $(17.3 - 14.6) = 2.7W$ deficit compared to Windows on DVD movie playback.

4.3.2 DVD Movie Single vs. Dual Core

DVD playback was measured with 1 CPU available vs. 2 CPUS, and there was zero impact on battery life.

4.3.3 DVD Movie: Throttling is not helpful

DVD playback was measured at T4 (50% throttling) and there was a net *negative* impact of 9 minutes on battery life. Again, throttling should be reserved for thermal management, and is almost never an appropriate tool where efficient performance/power is the goal.

4.4 Office Workload Battery Life and Performance

The Office workload battery life and performance are shown in Figure 11 and Figure 12, respectively. The example system lasted 232 minutes with `maxcpus=1` and a 5400 RPM drive, achieving a performance rating of 94 (UP5K in Figures 11 and 12). Enabling the second core cost 6 minutes (–3%) of battery life, but increased performance by 89% to to 178, (MP5K in Figures 11 and 12).

Upgrading the 5400 RPM disk drive to the 7200 RPM model had an 18 minute (8%) impact on the UP battery life, and an 12 minute (5%) impact on MP battery life. But the 7200 RPM drive had negligible performance benefit on this

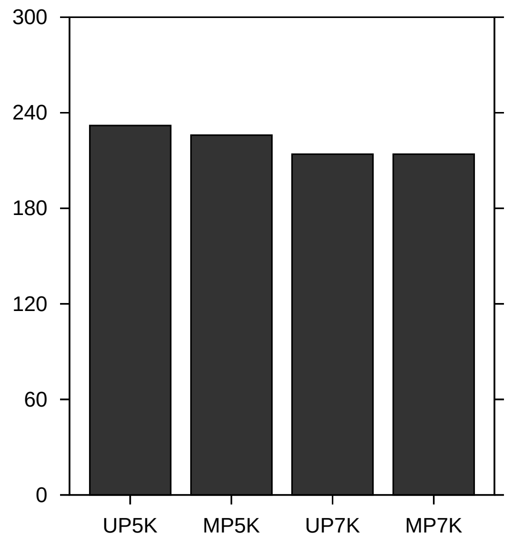


Figure 11: Office Battery Life

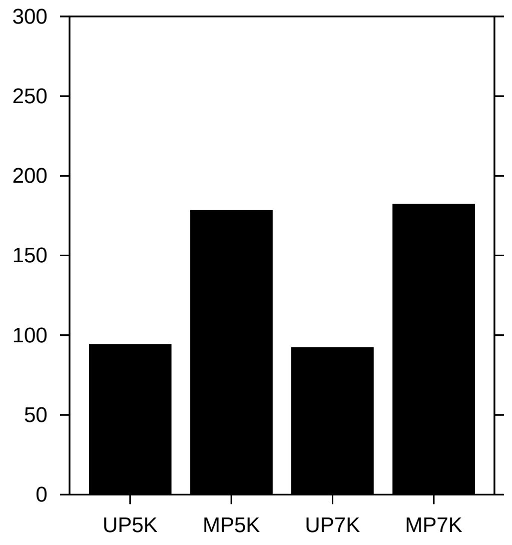


Figure 12: Office Performance

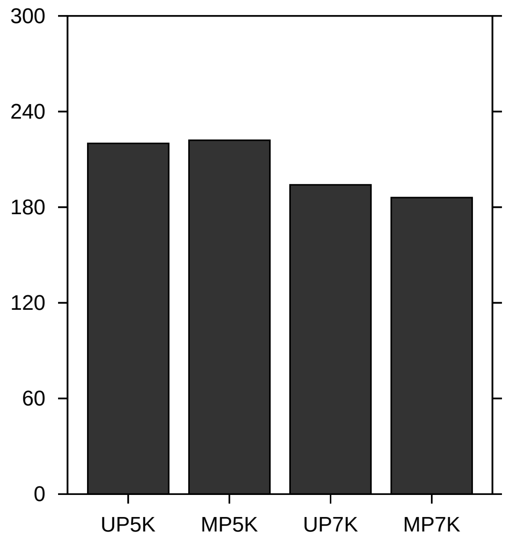


Figure 13: Developer Battery Life

workload. (UP7K and MP7K in Figures 11 and 12).

Note that the size of memory compared to the working set of the Office workload impact how much the disk is accessed. Were memory to be smaller or the workload modified to access the disk more, the faster drive would undoubtedly have a measurable benefit.

In summary, the second core has a significant performance benefit, with minimal battery cost on this workload. However, upgrading from a 5400RPM to 72000 RPM drive does not show a significant performance benefit on this workload as it is currently implemented.

4.5 Developer Workload Battery Life and Performance

The Developer workload battery life and performance are shown in Figure 13 and Figure 14, respectively.

Here the `maxcpus=1` 5400 RPM baseline scores 220 minutes with performance of 96.

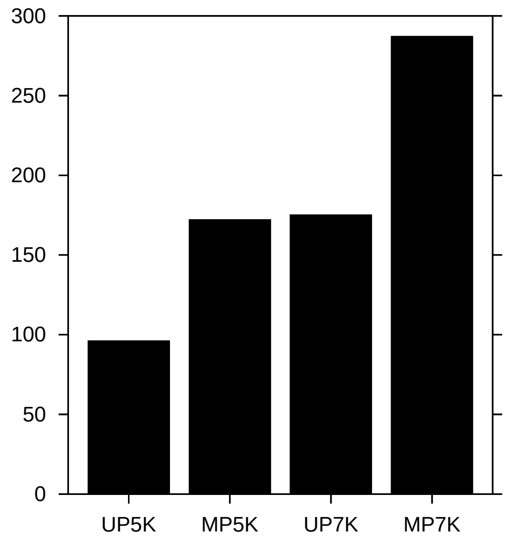


Figure 14: Developer Performance

Enabling the second core had a net positive impact on battery life of 2 minutes, and increased performance to 172 (+79%). Starting from the same baseline, upgrading to the 7200 RPM drive from the 5400 RPM drive dropped battery life 26 minutes to 194 from 220 (–12%), but increased performance to 175 (+82%). Simultaneously enabling the second core and upgrading the drive reduced battery life 34 minutes to 186 from 220 (15%), but increased performance to 287 (+198%).

Clearly developers using this class of machine should always have both cores enabled and should be using 7200 RPM drives.

5 Future Work

5.1 Enhancing the Tools

The current version of the tools, 1.0.4, could use some improvements.

- Concurrent Office applications. The current scripts start an application, use it, and then close it. Real users tend to have multiple applications open at once. It is unclear if this will have any significant effect on battery life, but it would be better eye candy.
- Add sanity checking that the system is properly configured before starting a measurement.

5.2 More Comparisons to make

The example measurements in this paper suggest even more measurements.

- Effect of run-time device power states.
- Comparison of default policies of different Linux distributors.
- Benefits of the laptop patch?
- USB 2.0 memory stick cost
- Gbit LAN linked vs unplugged
- WLAN seeking
- WLAN associated
- Bluetooth
- KDE vs. Gnome GUI
- LCD: brightness vs power consumption is there an optimal brightness/power setting?
- Power consumption while suspended to RAM vs. power consumption to reboot. What is break-even for length of time suspended vs halt, off, boot?
- Suspend to Disk and wakeup vs. staying idle
- Suspend to RAM and wakeup vs staying idle

6 Conclusion

The authors hope that the tools and techniques shown here will help the Linux community effectively analyze system power, understand laptop battery life, and improve Linux power management.

Appendix A: Scripting Commands

Keystrokes, keystroke conditions (like <Alt>, <Ctrl>, <Shift>, etc.) and delays are scripted in a scenario file along with other actions (`run command`, `wait command`, `select window`, `send signal`, etc.). The scenario file is passed to the workload script execution program, strings are parsed and appropriate actions are executed.

The scenario script is linear, no procedure defining is (currently) supported. Each string consists of 5 white space separated fields and begins with command name followed by 4 arguments (State, Count, Delay, String). For each particular command arguments could have different meanings or be meaningless, though all 4 arguments should present. The following commands are implemented:

Commands to generate user input

`DELAY 0 0 Delay 0`
: Suspends execution for 'Delay' msecs;

`PRESSKEY State Count Delay String`
: Send *Count* `State + String` keystrokes with *Delay* msec intervals between them to the window in focus, i.e. command `PRESSKEY S 2 500 Down` would generate two `<Shift> + <Down>` keystrokes with 1/2 second intervals. The state values are:

  S for Shift,

  A for Alt,

  C for Ctrl.

  Some keys should be presented as their respective names: Up, Down, Left, Right, Return, Tab, ESC.

`RELEASEKEY 0 0 0 String`
: Similar to `PRESSKEY` command, except the *Release* event being sent. It could be useful since some menu buttons react on key release, i.e. the pair of `PRESSKEY 0 0 <Delay> Return` and `RELEASEKEY 0 0 <Delay> Return` should be used in this case.

`TYPETEXT State 0 Delay String`
: Types text from *String* with *Delay* msecs interval between keystrokes. If *State* is `F` then the text from *String* file is typed instead of *String* itself.

`ENDSCEN 0 0 0 0`
: End of scenario. No strings beyond this one will be executed.

Commands to operate applications

`RUNCMD 0 0 0 String`
: Execute command *String*, exits on completion.

`WAITSTARTCMD 0 Count Delay String`
: Checks *Count* times with *Delay* msecs intervals if *String* command is started (total wait time is *Count* * *Delay* msecs).

`WAITFINISHCMD 0 Count Delay String`
: Checks *Count* times with *Delay* msecs intervals if *String* command is finished (total wait time is *Count* * *Delay* msecs).

Commands to interact with X windows

`SETWINDOWID State 0 0 String`
: Makes window with X window ID located in 'String' object active. If *State* is `F`, then *String* is treated as a file; if `E` or `0`, as an environment variable;

`SETWINDOW 0 0 0 String`
: Waits for window with *String* title to appear and makes it active.

`FOCUSIN 0 0 0 0`
: Sets focus to current active window.

`FOCUSOUT 0 0 0 0`
: Gets focus out of current active window.

`ENDWINDOW 0 0 0 String`
: Waits for window with *String* title to disappear.

`SYNCWINDOW 0 0 0 0`
: Tries to synchronize current active window.

To reach one particular window, `SETWINDOW` and `FOCUSIN` commands should be performed.

Commands to generate statistics

`SENDWORKMSG 0 0 0 String`
: Generate 'WORK' statistics string in log file with the *String* comment.

`SENDIDLEMSG 0 0 0 String`
: Generate 'IDLE' statistics string in log file with the *String* comment.

Note that the harness generates statistics regularly, so the above commands are intended to generate strings to mark the beginning and ending of the set of operations (e.g. 'hot-spot'), for which special measurements are required.

Debugging Commands

`TRACEON 0 0 0 0`
: Enable debug prints;

`TRACEOFF 0 0 0 0`
: Disable debug prints;

References

[ACPI] Hewlett-Packard, Intel, Microsoft, Phoenix, Toshiba *Advanced Configuration & Power Specification*, Revision 3.0a, December 30, 2005. `http://www.acpi.info`.

[Linux/ACPI] Linux/ACPI Project Home page, `http://acpi.sourceforge.net`.

[MM02] MobileMark® 2002, Business Applications Performance Corporation, `http://bapco.com`, June 4, 2002, Revision 1.0.

[MM05] MobileMark® 2005, Business Applications Performance Corporation, `http://bapco.com`, May 26, 2005, Revision 1.0.

BAPCo is a U.S. Registered Trademark of the Business Applications Performance Corporation. MobilMark is a U.S. Registered Trademark of the Business Applications Performance Corporation. Linux is a registered trademark of Linus Torvalds. All other trademarks mentioned herein are the property of their respective owners.

The Frysk Execution Analysis Architecture

Andrew Cagney
Red Hat Canada Limited
`cagney@redhat.com`

Abstract

The goal of the Frysk project is to create an intelligent, distributed, always-on system-monitoring and debugging tool. Frysk will allow GNU/Linux developers and system administrators: to monitor running processes and threads (including creation and destruction events); to monitor the use of locking primitives; to expose deadlocks, to gather data. Users debug any given process by either choosing it from a list or by accepting Frysk's offer to open a source code or other window on a process that is in the process of crashing or that has been misbehaving in certain user-definable ways.

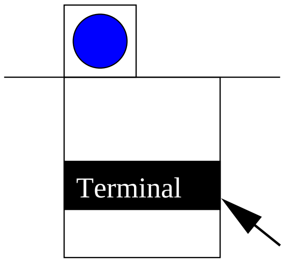
1 Introduction

This paper will first present a typical Frysk use-case. The use-case will then be used to illustrate how Frysk differs from a more traditional debugger, and how those differences benefit the user. This paper will then go on to provide a more detailed overview of Frysk's internal architecture and show how that architecture facilitates Frysk's objectives. Finally, Frysk's future development will be reviewed, drawing attention areas of the Linux Kernel that can be enhanced to better facilitate advanced debugging tools.

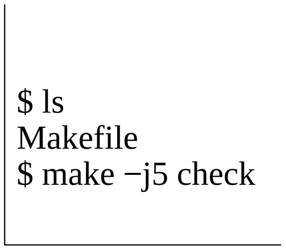
2 Example — K. the Compiler Engineer

K., a compiler engineer, spends a lot of time running a large, complex test-suite involving lots of processes and scripts, constantly monitoring each run for compiler crashes. When a crash occurs, K. must first attempt to reproduce the problem in isolation, then reproduce it under a command-line debugging tool, and then finally attempt to diagnose the problem.

Using Frysk, K. creates a monitored terminal:

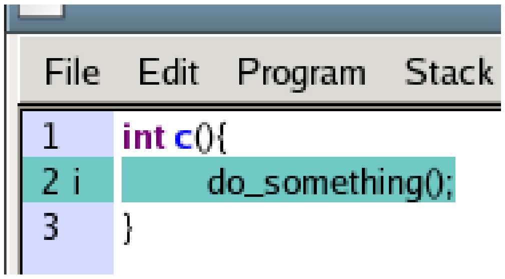


From within that terminal, K. can directly run the test framework:



```
$ ls
Makefile
$ make −j5 check
```

When a crash occurs, K. is alerted by the blinking Frysk icon in the toolbar. K. can then click on the Frysk icon and bring up the source window displaying the crashing program at the location at which the crash occurred:



```
int c(){
      do_something();
}
```

3 Frysk Compared To Traditional Debugger Technology

In the 1980s, at the time when debuggers such as GDB, SDB, and DBX were first developed, UNIX application complexity was typically limited to single-threaded, monolithic applications running on a single machine and written in C. Since that period, applications have grown both in complexity and sophistication utilizing: multiple threads and multiple processes; shared libraries; shared-memory; a distributed structure, typically as a client-server architecture; and implemented using C++, Java, C#, and scripting languages.

Unfortunately, the debugger tools developed at that time have failed to keep pace of these advances. Frysk, in contrast, has the goal of supporting these features from the outset.

3.1 Frysk Supports Multiple Threads, Processes, and Hosts

Given that even a simple application, such as firefox, involves both multiple processes and threads, Frysk was designed from the outset to follow Threads, Processes, and Hosts. That way the user, such as K., is provided with a single consistent tool that monitors the entire application.

3.2 Frysk is Non-stop

Historically, since an application had only a single thread, and since any sign of a problem (e.g., a signal) was assumed to herald disaster, the emphasis on debugging tools was to stop an application at the earliest sign of trouble. With modern languages, and their managed run-times, neither of those these assumptions apply. For instance, where previously a `SIGSEGV` was indicative of a fatal crash, it is now a normal part of an application's execution being used by the system's managed run-time as part of memory management.

With Frysk, the assumption is that the user requires the debugging tool to be as unobtrusive as possible, permitting the application to run freely. Only when the user explicitly requests control over one or more threads, or when a fatal situation such as that K. encountered is detected, will Frysk halt a thread or process.

3.3 Frysk is Event Driven

Unlike simpler command-line debugging tools, which are largely restricted to exclusively monitoring just the user's input or just the running application, Frysk is event-driven and able to co-ordinate both user and application events simultaneously. When implementing a graphical interface, this becomes especially important as the user expects Frysk to always be responsive.

3.4 Frysk has Transparent Attach and Detach

With a traditional debugging tool, a debugging session for an existing process takes the form:

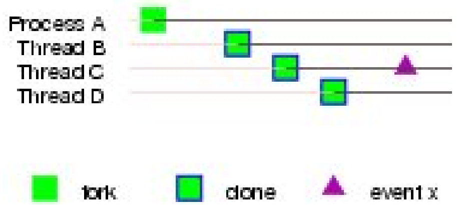
- attach to process
- examine values, continue, or stop
- detach from process

That is, the user is firstly very much aware of the state of the process (attached or detached), and secondly, is restricted to just manipulating attached processes. With Frysk, the user can initiate an operation at any time, the need to attach being handled transparently.

For instance, when a user requests a stack backtrace from a running process, Frysk automatically attaches to, and then stops, the process.

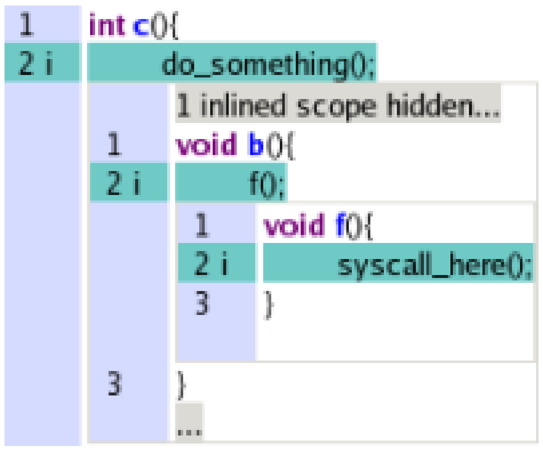
3.5 Frysk is Graphical, Visual

While a command-line based tool is useful for examining a simple single-threaded program, it is not so effective when examining an application that involves tens if not hundreds of threads. In contrast, Frysk strongly emphasizes its graphical interface providing visual mechanisms for examining an application. For instance, to examine the history of processes and events, Frysk provides an event line:



3.6 Frysk Handles Optimized and In-line Code

Rather than limiting debugging to applications that are written in C and compiled unoptimized, Frysk is focused on supporting application that have been compiled with optimized and in-lined code. Frysk exploits its graphical interface by permitting the user to examine the in-lined code in-place. For instance, an in-lined function `b()` with a further in-line call to `f()` can be displayed as:



```
1    int c(){
2 i      do_something();
          1 inlined scope hidden...
     1   void b(){
     2 i     f();
          1   void f(){
          2 i     syscall_here();
          3   }
     3   }
         ...
```

3.7 Frysk Loads Debug Information On-demand

Given that a modern application often has gigabytes of debug information, the traditional approach of reading all debug information into memory is not practical. Instead Frysk, using `libelf` and `libdw`, reads the debug information on demand, and hence ensures that Frysk's size is minimized.

3.8 Frysk Itself is Multi-Threaded and Object Oriented

It is often suggested that a debugging tool is best suited at debugging itself. This view being based on the assumption that since developers spend most of their time using their own tools for debugging their own tools, they will be strongly motivated to at least make debugging their tool easy. Consequently, a single-threaded procedural debugging tool written in C would be best suited for debugging C, while developers working on a multi-threaded, object-oriented, event-driven debugging tool are going to have a stronger motivation to make the tool work with that class of application.

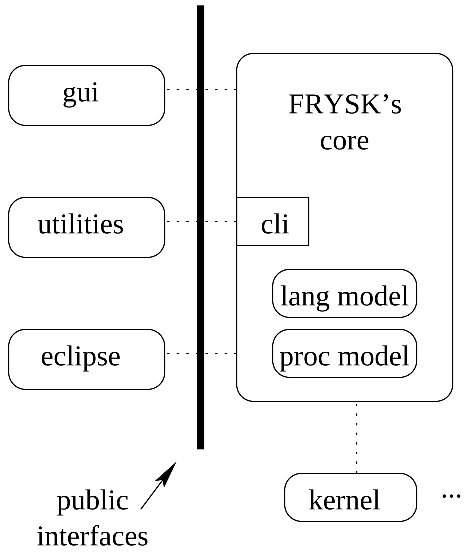
3.9 Frysk is Programmable

In addition to a graphical interface, the Frysk architecture facilitates the rapid development of useful standalone command-line utilities implemented using Frysk's core. For instance the command line utility `ftrace`, similar to `strace`, was implemented by adding a system call observer that prints call information to the threads being traced, and the program `fstack` was implemented by adding a stop observer to all threads of a process so that as each thread stopped its stack back-trace could be printed.

4 The Frysk Architecture

4.1 Overview

At a high level, Frysk's architecture can be viewed as a collection of clients that interact with Frysk's core. The core provides clients with alternate models or views of the system.



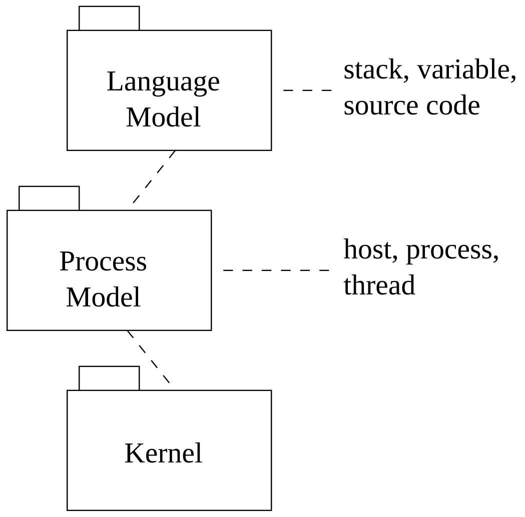
Frysk's core then uses the target system's kernel interfaces to maintain the internal models of the running system.

4.2 The Core, A Layered Architecture

Aspects of a Linux system can be viewed, or modeled, at different levels of abstraction. For instance:

- a process model: as a set of processes, each containing threads and each thread having registers and memory
- a language model: a process executing a high-level program, written in C++, having a stack, variables, and code

Conceptually, the models form sequence of layers, and each layer is implemented using the one below:

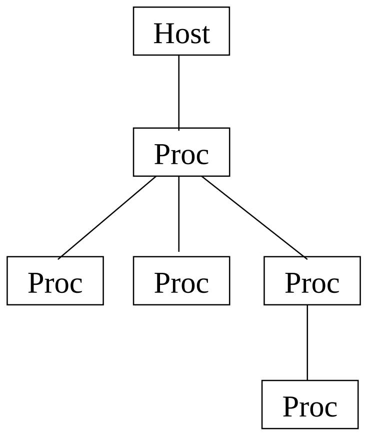


For instance, the language model, which abstracts a stack, would construct that stack's frames using register information obtained from the process model.

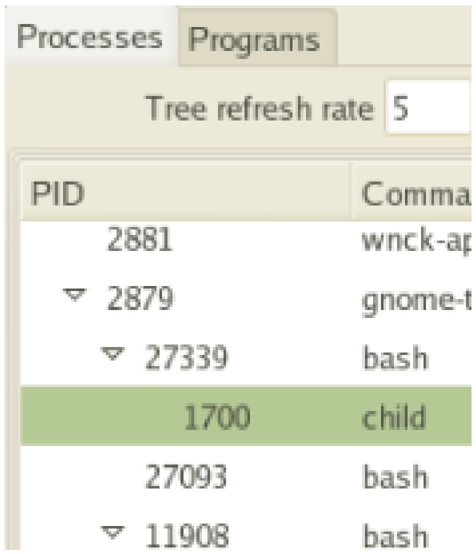
The core then makes each of those models available to client applications.

4.2.1 Frysk's Process Model

Frysk's process model implements a process-level view of the Linux system. The model consists of host, process, and task (or thread) objects corresponding to the Linux system equivalents:



Frysk then makes this model available to the user as part of the process window:

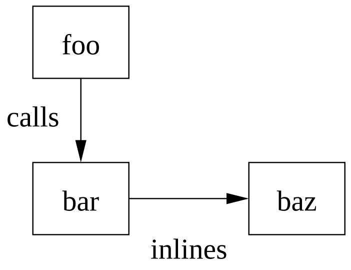


PID	Comma
2881	wnck-ap
2879	gnome-t
27339	bash
1700	child
27093	bash
11908	bash

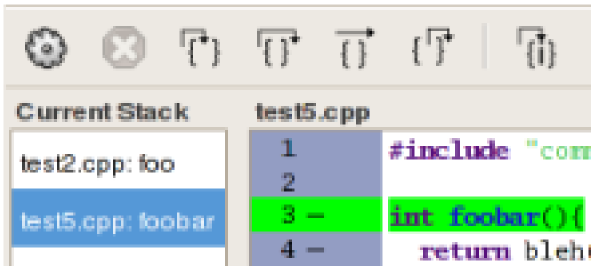
When a user requests that Frysk monitor for a process model event, such as a process exiting, that request is implemented by adding an observer (or monitor) to the objects to which the request applies. When the corresponding event occurs, the observers are notified.

4.2.2 Frysk's Language Model

Corresponding to the run-time state of a high-level program, Frysk provides a run-time language model. This model provides an abstraction of a running-program's stack (consisting of frames), variables and objects.



The model is then made available to the user through the source window's stack and source code browsers:



```
#include "com

int foobar(){
  return bleh
```

5 Future Direction

Going forward, Frysk's development is expected to be increasingly focused on large complex and distributed applications. Consequently Frysk is expected to continue pushing its available technology.

Internally, Frysk has already identified limitations of the current Linux Kernel debugging interfaces (`ptrace` and `proc`). For instance: that only the thread that did the attach be permitted to manipulate the debug target, or that waiting on kernel events still requires the juggling of `SIGCHLD` and `waitpid`. Addressing these issues will be critical to ensuring Frysk's scalability.

At the user level, Frysk will continue its exploration of interfaces that allow the user to analyze and debug increasingly large and distributed applications. For instance, Frysk's interface needs to be extended so that it is capable of visualizing and managing distributed applications involving hundreds or thousands of nodes.

6 Conclusion

Through the choice of a modern programming language, and the application of modern software design techniques, Frysk is well advanced in its goal of creating an intelligent, distributed, always-on monitoring and debugging tool.

7 Acknowledgments

Thanks goes to Michael Behm, Stan Cox, Adam Jocksch, Rick Moseley, Chris Moller, Phil Muldoon, Sami Wagiaalla, Elena Zannoni, and Wu Zhou, who provided feedback, code, and screenshots.

Evaluating Linux Kernel Crash Dumping Mechanisms

Fernando Luis Vázquez Cao
NTT Data Intellilink
`fernando@intellilink.co.jp`

Abstract

There have been several kernel crash dump capturing solutions available for Linux for some time now and one of them, kdump, has even made it into the mainline kernel.

But the mere fact of having such a feature does not necessary imply that we can obtain a dump reliably under any conditions. The LKDTT (Linux Kernel Dump Test Tool) project was created to evaluate crash dumping mechanisms in terms of success rate, accuracy and completeness.

A major goal of LKDTT is maximizing the coverage of the tests. For this purpose, LKDTT forces the system to crash by artificially recreating crash scenarios (panic, hang, exception, stack overflow, hang, etc.), taking into account the hardware conditions (such as ongoing DMA or interrupt state) and the load of the system. The latter being key for the significance and reproducibility of the tests.

Using LKDTT the author could constate the superior reliability of the kexec-based approach to crash dumping, although several deficiencies in kdump were revealed too. Since the final goal is having the best crash dumping mechanism possible, this paper also addresses how the aforementioned problems were identified and solved. Finally, possible applications of kdump beyond crash dumping will be introduced.

1 Introduction

Mainstream Linux lacked a kernel crash dumping mechanism for a long time despite the fact that there were several solutions (such as *Diskdump* [1], *Netdump* [2], and *LKCD* [3]) available out of tree . Concerns about their intrusiveness and reliability prevented them from making it into the vanilla kernel.

Eventually, a handful of crash dumping solutions based on *kexec* [4, 5] appeared: *Kdump* [6, 7], *Mini Kernel Dump* [8], and *Tough Dump* [9]. On paper, the kexec-based approach seemed very reliable and the impact in the kernel code was certainly small. Thus, kdump was eventually proposed as Linux kernel's crash dumping mechanism and subsequently accepted.

However, having a crash dumping mechanism does not necessarily imply that we can get a dump under any crash scenario. It is necessary to do proper testing, so that the success rate and accuracy of the dumps can be estimated and the different solutions compared fairly. Besides, having a standardised test suite would also help establishing a quality standard and, collaterally, detecting regressions would be much easier.

Unless otherwise indicated, henceforth all the explanations will refer to i386 and x86_64 architectures, and Linux 2.6.16 kernel.

1.1 Shortcomings of current testing methods

Typically to test crash dumping mechanisms a kernel module is created that artificially causes the system to die. Common methods to bring the system down from this module consist of directly invoking `panic`, making a null pointer dereference and other similar techniques.

Sometimes, to ease testing a user space tool is provided that sends commands to the kernel-space part of the testing tool (via the `/proc` file system or a new device file), so that things like the crash type to be generated can be configured at run-time.

Beyond the crash type, there are no provisions to further define the crash scenario to be recreated. In other words, parameters like the load of the machine and the state of the hardware are undefined at the time of testing.

Judging from the results obtained with this approach to testing all crash dumping solutions seem to be very close in terms of reliability, regardless of whether they are kexec-based or not, which seems to contradict theory. The reason is that the coverage of the tests is too limited as a consequence of leaving important factors out of the picture. Just to give some examples, the hardware conditions (such as ongoing DMA or interrupt state), the system load, and the execution context are not taken into consideration. This greatly diminishes the relevance of the results.

1.2 LKDTT motivation

The critical role crash dumping solutions play in enterprise systems calls for proper testing, so that we can have an estimate of their success rate under realistic crash scenarios. This is something the current testing methods cannot achieve and, as an attempt to fill this gap, the LKDTT project [10] was created.

Using LKDTT many deficiencies in kdump, LKCD, mkdump and other similar projects were found. Over the time, some regressions were observed too. This type of information is of great importance to both Linux distributions and end-users, and making sure it does not pass unnoticed is one of the commitments of this project.

To create meaningful tests it is necessary to understand the basics of the different crash dumping mechanisms. A brief introduction follows in the next section.

2 Crash dump

A variety of crash dumping solutions have been developed for Linux and other UNIX®-like operating systems over the time. Even though implementations and design principles may differ greatly, all crash dumping mechanisms share a multistage nature:

1. Crash detection.
2. Minimal machine shutdown.
3. Crash dump capture.

2.1 Crash detection

For the crash dump capturing process to start a trigger is needed. And this trigger is, most interestingly, a system crash.

The problem is that this peculiar trigger sometimes passes unnoticed or, in the words, *the kernel is unable to detect that itself has crashed.*

The culprits of system crashes are *software errors* and *hardware errors*. Often a hardware error leads to a software errors, and vice versa, so it is not always easy to identify the original problem. For example, behind a panic in the VFS code a damaged memory module might be lurking.

There is one principle that applies to both software and hardware errors: if the intention is to capture a dump, as soon as an error is detected control of the system should be handed to the crash dumping functionality. Deferring the crash dumping process by delegating invocation of the dump mechanism to functions such as `panic` is potentially fatal, because the crashing kernel might well lose control of the system completely before getting there (due to a stack overflow for example).

As one might expect, the detection stage of the crash dumping process does not show marked implementation specific differences. As a consequence, a single implementation could be easily shared by the different crash dumping solutions.

2.1.1 Software errors

A list of the most common crash scenarios the kernel has to deal with is provided below:

- *Oops*: Occurs when a programming mistake or an unexpected event causes a situation that the kernel deems grave. Since the kernel is the supervisor of the entire system it cannot simply kill itself as it would do with a user-space application that goes nuts. Instead, the kernel issues and oops (which results in a stack trace and error message to the console) and strives to get out of the situation. But often, after the oops, the system is left in an inconsistent state the kernel cannot recover from and, to avoid further damage, the system panics (see panic below). For example, a driver might have been in the middle of talking to hardware or holding a lock at the time of the crash and it would not be safe to resume execution. Hence, a panic is issued instead.
- *Panic*: Panics are issued by the kernel upon detecting a critical error from which it cannot recover. After printing and error message the system is halted.
- *Faults*: Faults are triggered by instructions that cannot or should not be executed by the CPU. Even though some of them are perfectly valid, and in fact play an essential role in important parts of the kernel (for example, pages faults in virtual memory management); there are certain faults caused by programming errors, such as divide-error, invalid TSS, or double fault (see below), which the kernel cannot recover from.
- *Double and triple faults*: A double fault indicates that the processor detected a second exception while calling the handler for a previous exception. This might seem a rare event but it is possible. For example, if the invocation of an exception handler causes a stack overflow a page fault is likely to happen, which, in turn, would cause a double fault. In i386 architectures, if the CPU faults again during the inception of the double fault, then it triple faults, entering a shutdown cycle that is followed by a system `RESET`.
- *Hangs*: Bugs that cause the kernel to loop in kernel mode, without giving other tasks the chance to run. Hangs can be classified in two big groups:
  - *Soft lockups*: These are transitory lockups that delay execution and

scheduling of other tasks. Soft lockups can be detected using a software watchdog.

  - *Hard lockups*: These are lockups that leave the system completely unresponsive. They occur, for example, when a CPU disables interrupts and gets stuck trying to get spinlock that is not freed due to a locking error. In such a state timer interrupts are not served, so scheduler-based software watchdogs cannot be used for detection. The same happens to keyboard interrupts, and that is why the `Sys Rq` key cannot be used to trigger the crash dump. The solution here is using the NMI handler.

- *Stack overflows*: In Linux the size of the stacks is limited (at the time of writing i386's default size is 8KB) and, for this reason, the kernel has to make a sensitive use of the stack to avoid bloating. It is a common mistake by novice kernel programmers to declare large automatic variables or to use deeply nested recursive algorithms; both of these practises tend to cause stack overflows. Stacks are also jeopardised by other factors that are not so evident. For example, in i386 interrupts and exceptions use the stack of the current task, which puts extra pressure on it. Consequently, interruption nesting should also be taken into account when programming interrupt handlers.

2.1.2 Hardware errors

Not only software has errors, sometimes machines fail too. Some hardware errors are recoverable, but when a fatal error occurs the system should come to a halt as fast as possible to avoid further damage. It is not even clear whether trying to capture a crash dump in the event of a serious hardware error is a sensitive thing to do. When the underlying hardware cannot be trusted one would rather bring the system down to avoid greater havoc.

The Linux kernel can make use of some error detection facilities of computer hardware. Currently the kernel is furnished with several infrastructures which deal with hardware errors, although the tendency seems to be to converge around EDAC (Error Detection and Correction) [11]. Common hardware errors the Linux kernel knows about include:

- *Machine checks*: Machine checks occur in response to CPU-internal malfunctions or as a consequence of hardware resets. Their occurrence is unpredictable and can leave memory and/or registers in a partially updated state. In particular, the state of the registers at the time of the event is completely undefined.

- *System RAM errors*: In systems equipped with ECC memory the memory chip has extra circuitry that can detect errors in the ingoing and outgoing data flows.

- *PCI bus transfer errors*: The data travelling to/from a PCI device may experience corruption whilst on the PCI bus. Even though a majority of PCI bridges and peripherals support such error detection most system do not check for them. It is worth noting that despite the fact that some of this errors might trigger an NMI it is not possible figure out what caused it, because there is no more information.

2.2 Minimal machine shutdown

When the kernel finds itself in a critical situation it cannot recover from, it should hand control of the machine to the crash dumping functionality. In contrast to the previous stage (detection), the things that need to be done at this point are quite implementation dependent. That said, all the crash dumping solutions, regardless of their design principles, follow the basic execution flow indicated below:

1. Right after entering the dump route the crashing CPU disables interrupts and saves its context in a memory area specially reserved for that purpose.
2. In SMP environments, the crashing CPU sends NMI IPIs to other CPUs to halt them.
3. In SMP environments, each IPI receiving processor disables interrupts and saves its context in a special-purpose area. After this, the processor busy loops until the dump process ends.
   Note: Some kexec-based crash dump capturing mechanisms relocate to boot CPU after a crash, so this step becomes different in those cases (see Section 7.4 for details).
4. The crashing CPU waits a certain amount of time for IPIs to be processed by the other CPUs, if any, and resumes execution.
5. Device shutdown/reinitialization, if done at all, is kept to a minimum, for it is not safe after a crash.
6. Jump into the crash dump capturing code.

2.3 Crash dump capture

Once the minimal machine shutdown is completed the system jumps into the crash dump capturing code, which takes control of the system to do the dirty work of capturing the dump and saving the dump image in a safe place.

Before continuing, it is probably worth defining what is understood by *kernel crash dump*. A kernel crash dump is an image of the resources in use by the kernel at the time of the crash, and whose analysis is an essential element to clarify what went wrong. This usually comprises an image of the memory available to the crashed kernel and the register states of all the processors in the system. Essentially, any information deemed useful to figure out the source of the problem is susceptible of being included in the dump image.

This final and decisive stage is probably the one that varies more between implementations. Attending to the design principles two big groups can be identified though:

- *In-kernel solutions*: LKCD, Diskdump, Netdump.
- *kexec-based solutions*: Kdump, MK-Dump, Tough Dump.

2.3.1 In-kernel solutions

The main characteristic of the in-kernel approach is, as the name suggests, that the crash dumping code uses the resources of the crashing kernel. Hence, these mechanisms, among other things, make use of the drivers and the memory of the crashing kernel to capture the crash dump. As might be expected this approach has many reliability issues (see Section 5.1 for an explanation and Table 2 for test results).

2.3.2 kexec-based solutions

The core design principle behind the kexec-based approach is that the dump is captured from an independent kernel (the crash dump kernel or second kernel) that is soft-booted after the crash. Here onwards, for discussion purposes, crashing kernel is referred to as first kernel and the kernel which captures the dump as either capture kernel or second kernel.

As a general rule, the crash dumping mechanism should avoid fiddling with the resources (such as memory and CPU registers) of the crashing kernel. And, when this is inevitable, the state of these resources should be saved before they are used. This means that if the capture kernel wants to use a memory region the first kernel was using, it should first save the original contents so that this information is not missing in the dump image. These considerations along with the fact that we are soft booting into a new kernel determines the possible implementations of the capture kernel:

- *Booting the capture kernel from the standard memory location*
  The capture kernel is loaded in a reserved memory region and in the event of a crash it is copied to the memory area from where it will boot. Since this kernel is linked against the architecture's default start address, it needs to reside in the same place in memory as the crashing kernel. Therefore, the memory area necessary to accommodate the capture kernel is preserved by copying it to a backup region just before doing the copy. This was the approach taken by Tough Dump.

- *Booting the second kernel from a reserved memory region*
  After a crash the system is unstable and the data structures and functions of the crashing kernel are not reliable. For this reason there is no attempt to perform any kind of device shutdown and, as a consequence, any ongoing DMAs at the time of the crash are not stopped. If the approach discussed before is used the capture kernel is prone to be stomped by DMA transactions initiated in the first kernel. As long as IOMMU entries are not reassigned, this problem can be solved by booting the second kernel directly from the reserved memory region it was loaded into. To be able to boot from the reserved memory region the kernel has to be relocated there. The relocation can be accomplished at compile time (with the linker) or at run-time (see discussion in Section 7.2.1). Kdump and mkdump take the first and second approach, respectively.

For the reliability reasons mentioned above the second approach is considered the right solution.

3 LKDTT

3.1 Outline

LKDTT is a test suite that forces the kernel to crash by artificially recreating realistic crash scenarios. LKDTT accomplishes this by taking into account both the state of the hardware (for example, execution context and DMA state) and the load conditions of the system for the tests.

LKDTT has kernel-space and user-space components. It consists of a kernel patch that implements the core functionality, a small utility to control the testing process (`ttutils`), and a set of auxiliary tools that help recreating the necessary conditions for the tests.

Usually tests proceed as follows:

- If it was not built into the kernel, load the DTT (Dump Test Tool) module (see Section 3.2.2).

- Indicate the point in the kernel where the crash is to be generated using `ttutils` (see Section 3.3). This point is called Crash Point (CP).

- Reproduce the necessary conditions for the test using the auxiliary tools (see Section 3.4).

- Configure the CP using `ttutils`. The most important configuration item is the crash type.

- If the CP is located in a piece of code rarely executed by the kernel, it may become necessary to use some of the auxiliary tools again to direct the kernel towards the CP.

A typical LKDTT session is depicted in Table 1.

3.2 Implementation

LKDTT is pretty simple and specialized on testing kernel crash dumping solutions. LKDTT's implementation is sketched in Figure 1, which will be used throughout this section for explanation purposes.

The sequence of events that lead to an artificial crash is summarized below:

- Kernel execution flow reaches a crash point or CP (see 3.2.1). In the picture the CP is called `HD_CP` and is located in the hard disk device driver.

- If the CP is enabled the kernel jumps into the DTT module. Otherwise execution resumes from the instruction immediately after the CP.

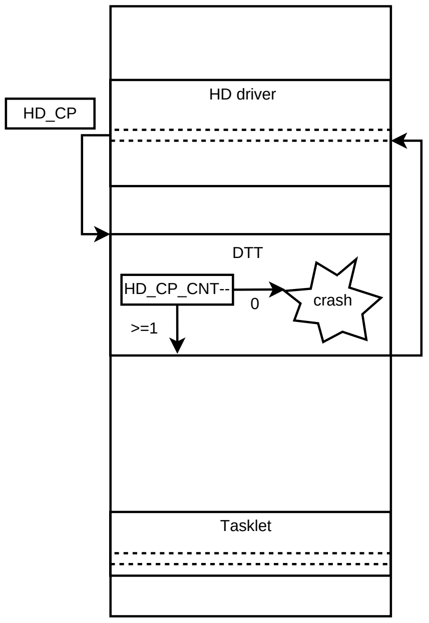


Figure 1: Crash points implementation

- The DTT module checks the counter associated with the CP. This counter (`HD_CP_CNT` in the example) keeps track of the number of times the CP in question has been crossed.

- This counter is a reverse counter in reality, and when it reaches 0 the DTT (Dump Test Tool) module induces the system crash associated with the CP. If the counter is still greater than zero execution returns from the module and continues from the instruction right after the CP.

Both the initial value of the counter and the crash type to be generated are run-time configurable from user space using `ttutils` (see 3.3). Crash points, however, are inserted in the kernel source code as explained in the following section.

```
# modprobe dtt
# ./ttutils ls
id      crash type      crash point name        count   location
1       none            INT_HARDWARE_ENTRY      0       kern
3       none            FS_DEVRW                0       kern
4       panic           MEM_SWAPOUT             7       kern
5       none            TASKLET                 0       kern
# ./ttutils add -p IDE_CORE_CP -n 50
# ./ttutils ls
id      crash type      crash point name        count   location
1       none            INT_HARDWARE_ENTRY      0       kern
3       none            FS_DEVRW                0       kern
4       panic           MEM_SWAPOUT             7       kern
5       none            TASKLET                 0       kern
50      none            IDE_CORE_CP             0       dyn
# ./helper/memdrain
# ./ttutils set -p IDE_CORE_CP -t panic -c 10
```

Table 1: LKDTT usage example

3.2.1 Crash Points

Each of the crash scenarios covered by the test suite is generated at a Crash Point (CP), which is a mere hook in the kernel. There are two different approaches to inserting hooks at arbitrary points in the kernel: patching the kernel source and dynamic probing. At first glance, the latter may seem the clear choice, because it is more flexible and it would not be necessary to recompile the kernel to insert a new CP. Besides, there is already an implementation of dynamic probing in the kernel (Kprobes [12]), with which the need of a kernel recompilation disappears completely.

Despite all these advantages dynamic probing was discarded because it changes the execution mode of the processor (a breakpoint interrupt is used) in a way that can modify the result of a test. Using `/dev/mem` to crash the kernel is another option, but in this case there is no obvious way of carrying out the tests in a controlled manner. These are the main motives behind LKDTT's election of the kernel patch approach. Specifically, the current CP implementation is based on IBM's *Kernel Hooks*.

In any case, a recent extension to Kprobes called Djprobe [13] that uses the breakpoint trap just once to insert a jump instruction at the desired probe point and uses this thereafter looks promising (the jump instruction does not alter the CPU's execution mode and, consequently, should not alter the test results).

As pointed out before, each CP has two attributes: number of times the CP is crossed before causing the system crash and the crash type to be generated.

At the time of writing, 5 crash types are supported:

- Oops: generates a kernel oops.
- Panic: generates a kernel panic.
- Exception: dereferences a null pointer.
- Hang: simulates a locking error by busy looping.

- Overflow: bloats the stack.

As mentioned before, crash points are inserted in the kernel source code. This is done using the macro `CPOINT` provided by LKDTT's kernel patch (see 3.5). An example can be seen in code listing 1.

3.2.2 DTT module

The core of LKDTT is the DTT (Dump Test Tool) kernel module. Its main duties are: managing the state of the CPs, interfacing with the user-space half of LKDTT (i.e. `ttutils`) through the `/proc` file system, and generating the crashes configured by the user using the aforementioned interface.

Figure 2 shows the pseudo state diagram of a CP (the square boxes identify states). From LKDTT's point of view, CPs come to existence when they are registered. This is done automatically in the case of CPs compiled into the kernel image. CPs residing in kernel modules, on the other hand, have to be registered either by calling a CP registration function from the module's init method, or from user space using a special `ttutils`' option (see `add` in *ttutils*, Section 3.3, for a brief explanation and Table 1 for a usage example). The later mechanism is aimed at reducing the intrusiveness of LKDTT.

Once a CP has been successfully registered the user can proceed to configure it using `ttutils` (see `set` in *ttutils*, Section 3.3, and Table 1 for an example). When the CP is enabled, every time the CP is crossed the DTT module decreases the counter associated with it by one and, when it reaches 0, LKDTT simulates the corresponding failure.

If it is an unrecoverable failure, the crash dumping mechanism should assume control of the system and capture a crash dump. However, if the kernel manages to recover from this eventuality the CP is marked as disabled, and remains in this state until it is configured again or deleted. There is a caveat here though: in-kernel CPs cannot be deleted.

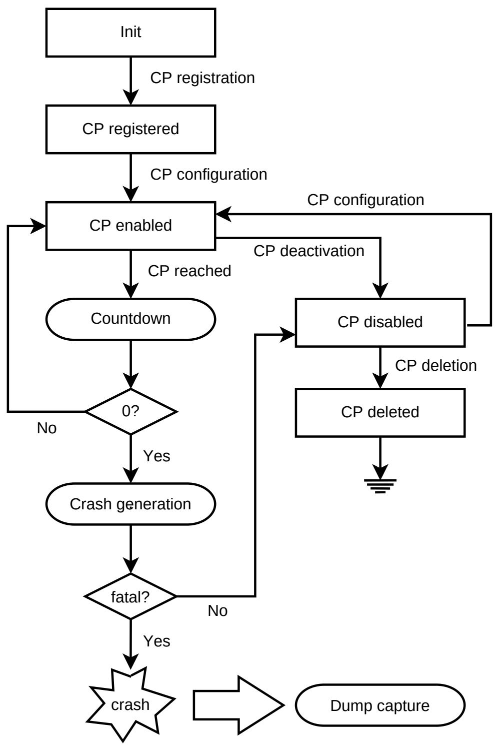


Figure 2: State diagram of crash points

LKDTT can be enabled either as a kernel module or compiled into the kernel. In the modular case it has to be modprobed:

```
# modprobe dtt [rec_num={>0}]
```

`rec_num` sets the recursion level for the stack overflow test (default is 10). The stack growth is approximately `rec_num*1KB`.

3.3 ttutils

`ttutils` is the user-space bit of LKDTT. It is a simple C program that interfaces with the

DTT module through `/proc/dtt/ctrl` and `/proc/dtt/cpoints`. The first file is used to send commands to the DTT module, commonly to modify the state of a CP. The second file, on the other hand, can be used to retrieve the state of crash points. It should be fairly easy to integrate the command in scripts to automate the testing process.

The ttutils command has the following format:

```
ttutils command [options]
```

The possible commands being:

- `help`: Display usage information.
- `ver(sion)`: Display version number of LKDTT.
- `list|ls`: Show registered crash points.
- `set`: Configure a Crash Point (CP). `set` can take two different options:
  `-p cpoint_name`: CP's name.
  `-t cpoint_type`: CP's crash type. Currently the available crash types are: *none* (do nothing), *panic*, *bug* (oops), *exception* (generates an invalid exception), *loop* (simulates a hang), and *overflow*.
  `-c pass_num`: Number of times the crash point has to be crossed before the failure associated with its type is induced. The default value for `pass_num` is 10.
- `reset`: Disable a CP. Besides, the associated counter that keeps track of the number of times the crash point has been traversed is also reset. The options available are:
  `-p cpoint_name`: CP's name.
  `-f`: Reset not only the CP's counter but also revert its type to none.
- `add`: Register a CP from a kernel module so that it can be actually used. This is aimed at modules that do not register the CPs inserted in their own code. Please note that registering does not imply activation. Activation is accomplished using `set`. `add` has two options:
  `-p cpoint_name`: CP's name.
  `-n id`: ID to be associated with the CP.
- `rmv`: Remove a CP registered using `add`. `rmv` has one single option:
  `-p cpoint_name`: CP's name.

3.4 Auxiliary tools

One of the main problems that arises when testing crash dumping solutions is that artfully inserting crash points in the kernel does not always suffice to recreate certain crash scenarios. Some execution paths are rarely trodden and the kernel has to be lured to take the right wrong way.

Besides, the tester may want the system to be in a particular state (ongoing DMA or certain memory and CPU usage levels, for example).

This is when the set of auxiliary tools included in LKDTT comes into play to reproduce the desired additional conditions.

A trivial example is `memdrain` (see Table 1 for a usage example), a small tool included in the LKDTT bundle. `memdrain` is a simple C program that reserves huge amounts of memory so that swapping is initiated. By doing so the kernel is forced to traverse the CPs inserted in the paging code, so that we can see how the crash dumping functionality behaves when a crash occurs in the middle of paging anonymous memory.

3.5 Installation

First, the kernel patch has to be applied:

```
# cd <PATH_TO_KERNEL_X.Y.Z>
# zcat <PATH_TO_PATCH>/dtt-full-X.
Y.Z.patch.gz | patch -p1
```

Once the patch has been applied we can proceed to configure the kernel as usual, but making sure that we select the options indicated below:

```
# make menuconfig
  Kernel hacking --->
   Kernel debugging [*]
    Kernel Hook support [*] or [M]
     Crash points [*] or [M]
```

The final steps consist of compiling the kernel and rebooting the system:

```
# make
# make modules_install
# make install
# shutdown -r now
```

The user space tools (`ttutils` and the auxiliary tools) can be installed as follows:

```
# tar xzf dtt_tools.tar.gz
# cd dtt_tools
# make
```

4 Test results

The results of some tests carried out with LKDTT against LKCD and two different versions of the vanilla kernel with kdump enabled can be seen in Table 2.

For each crash point all the crash types supported by LKDTT were tried: *oops*, *panic*, *exception*, *hang*, and *overflow*. The meaning of the crash points used during the tests is explained in Section 3.2.1.

The specifications of the test machine are as follows:

- CPU type: Intel Pentium 4 Xeon Hyperthreading.
- Number of CPUs: 2.
- Memory: 1GB.
- Disk controller: ICH5 Serial ATA.

The kernel was compiled with the options below turned on (when available): `CONFIG_PREEMPT`, `CONFIG_PREEMPT_BKL`, `CONFIG_DETECT_SOFTLOCKUP`, `CONFIG_4KSTACKS`, `CONFIG_SMP`. And the kernel command line for the kdump tests was:

```
root=/dev/sda1 ro crashkernel=32M@
16M nmi_watchdog=1 console=ttyS0,
38400 console=tty0
```

The test results in Table 2 are indicated using the convention below:

- `O`: Success.
- `O(nrbt)`: The system recovered from the induced failure. However, a subsequent reboot attempt failed, leaving the machine hanged.
- `O(nrbt, nmiw)`: The dump image was captured successfully but the system hanged when trying to reboot. The NMI watchdog detected this hang and, after determining that the crash dump had already been taken, tried to reboot the system again.

Crash point	Crash type	LKCD 6.1.0	kdump 2.6.13-rc7	kdump 2.6.16
`INT_HARDWARE_ENTRY`	panic	X	0	0
	oops	X (nmiw, nrbt)	X (nmiw)	0
	exception	X (nmiw, nrbt)	X (nmiw)	0
	hang	X	0	0
	overflow	X	X	X
`INT_HW_IRQ_EN`	panic	X	0	0
	oops	X	0	X(2c)
	exception	X	0	0
	hang	X	X	X
	overflow	X	X	X
`INT_TASKLET_ENTRY`	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0
	exception	0(nrbt, nmiw)	0	0
	hang	X	X (SysRq)	X(det,SysRq)
	overflow	X	X	X
`TASKLET`	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0
	exception	0(nrbt, nmiw)	0	0
	hang	0(nrbt, nmiw)	0	0
	overflow	X	X	X
`FS_DEVRW`	panic	0(nrbt, nmiw)	0	X(2c)
	oops	0(nrbt, nmiw)	0	X (log,SysRq)
	exception	0(nrbt, nmiw)	0	X (log,SysRq)
	hang	X	X (SysRq)	X (SysRq)
	overflow	X	X	X
`MEM_SWAPOUT`	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0 (nrbt)
	exception	0(nrbt, nmiw)	0	0 (nrbt)
	hang	X	X	X (unk,SysRq,2c)
	overflow	X	X	X
`TIMERADD`	panic	0(nrbt, nmiw)	0	0
	oops	0(nrbt, nmiw)	0	0
	exception	0(nrbt, nmiw)	0	0
	hang	X	0	0
	overflow	X	X	X
`SCSI_DISPATCH_CMD`	panic	X	0	0
	oops	X	0	0
	exception	X	0	0
	hang	X	X (SysRq)	X (det,SysRq)
	overflow	X	X	X

Table 2: LKDTT results

- `X`: Failed to capture dump.

- `X(2c)`: After the crash control of the system was handed to the capture kernel, but it crashed due to a device initialization problem.

- `X(SysRq)`: The crash not detected by the kernel, but the dump process was successfully started using the `Sys Rq` key.
Note: Often, when plugged after the crash the keyboard does not work and the `Sys Rq` is not effective as a trigger for the dump.

- `X(SysRq, 2c)`: Like the previous case, but the capture kernel crashed trying to initialize a device.

- `X(det, SysRq)`: The hang was detected by the soft lockup watchdog (`CONFIG_DETECT_SOFTLOCKUP`). Since this watchdog only notifies about the lockup without taking any active measures the dump process had to be started using the `Sys Rq` key.
Note: Even though the dump was successfully captured the result was marked with an X because it required user intervention. The action to take upon lockup should be configurable.

- `X(log, SysRq)`: The system became increasingly unstable, eventually becoming impossible to login into the system anymore (the prompt did not return after introducing login name and password). After the system locked up like this, the dump process had to be initiated using the `Sys Rq` key, because neither the NMI watchdog nor the soft lockup watchdog could detect any anomaly.

- `X(nmiw)`: The error was detected but the crashing kernel failed to hand control of the system to the crash dumping mechanism and hanged. This hang was subsequently detected by the NMI watchdog, who succeed in invoking the crash dumping functionality. Finally, the dump was successfully captured.
Note: The result was marked with an X because the NMI watchdog sometimes fails to start the crash dumping process.

- `X(nmiw, nrbt)`: Like the previous case, but after capturing the dump the system hanged trying to reboot.

- `X(unk,SysRq,2c)`: The auxiliary tool used for the test (see Section 3.4) became unkillable. After triggering the dump process using the `Sys Rq` key, the capture kernel crashed attempting to reinitialize a device.

4.1 Crash points

Even though testers are free to add new CPs, LKDTT is furnished with a set of essential CPs, that is, crash scenarios considered basic and that should always be tested. The list follows:

IRQ handling with IRQs disabled (`INT_HARDWARE_ENTRY`) This CP is crossed whenever an interrupt that is to be handled with IRQs disabled occurs (see code listing 1).

IRQ handling with IRQs enabled (`INT_HW_IRQ_EN`) This is the equivalent to the previous CP with interrupts enabled (see code listing 2).

Tasklet with IRQs disabled (`TASKLET`) If this CP is active crashes during the service of Linux tasklets with interrupts disabled can be recreated.

```
fastcall unsigned int __do_IRQ(
   unsigned int irq, struct
   pt_regs *regs)
{
  .....

  CPOINT(INT_HARDWARE_ENTRY);
  for (;;) {
    irqreturn_t action_ret;

    spin_unlock(&desc->lock);

    action_ret = handle_IRQ_event(
       irq, regs, action);
  .....
}
```

Listing 1: `INT_HARDWARE_ENTRY` crash point (`kernel/irq/handle.c`)

```
fastcall int handle_IRQ_event(
  .....

  if (!(action->flags &
     SA_INTERRUPT)) {
    local_irq_enable();
    CPOINT(INT_HW_IRQ_EN);
  }

  do {
    ret = action->handler(irq,
       action->dev_id, regs);
    if (ret == IRQ_HANDLED)
      status |= action->flags;
  .....
}
```

Listing 2: `INT_HW_IRQ_EN` crash point (`kernel/irq/handle.c`)

Tasklet with IRQs enabled (`INT_TASKLET_ENTRY`) Using this CP it is possible to cause a crash when the kernel is in the middle of processing a tasklet with interrupts enabled.

Block I/O (`FS_DEVRW`) This CP is used to bring down the system while the file system is performing low-level access to block devices (see code listing 3).

Swap-out (`MEM_SWAPOUT`) This CP is located in the code that tries to allocate space for anonymous process memory.

Timer processing (`TIMERADD`) This is a CP situated in the code that starts and re-starts high resolution timers.

SCSI command (`SCSI_DISPATCH_CMD`) This CP is situated in the SCSI command dispatching code.

IDE command (`IDE_CORE_CP`) This CP is situated in the code that handles I/O operations on IDE block devices.

5 Interpretation of the results and possible improvements

5.1 In-kernel crash dumping mechanisms (LKCD)

The primary cause of the bad results obtained by LKCD, and in-kernel crash dumping mechanism in general, is the flawed assumption that the kernel can be trusted and will in fact be operating in a normal fashion. This creates two major problems.

First, there is a problem with resources, notably with resources locking up, because it is not possible to know the locking status at the time of the crash. LKCD uses drivers and services of the crashing kernel to capture the dump. As a

```
void ll_rw_block(int rw, int nr,
   struct buffer_head *bhs[])
{
  .....
        get_bh(bh);
        submit_bh(rw, bh);
        continue;
      }
    }
    unlock_buffer(bh);
    CPOINT(FS_DEVRW);
  }
}
```

Listing 3: `FS_DEVRW` crash point (`fs/buffer.c`)

consequence, if the operation that has caused the crash was locking resources necessary to capture the dump, the dump operation will end up deadlocking. For example, the driver for the dump device may try to obtain a lock that was held before the crash occurred and, as it will never be released, the dump operation will hang up. Similarly, on SMP systems as operations being run on other CPUs are forced to stop in the event of a crash, there is the possibility that resources needed during the dumping process may be locked, because they were in use by any of the other CPUs and were not released before they halted. This may put the dump operation into a lockup too. Even if this doesn't result in a lock-up, insufficient system resources may also cause the dump operation to fail.

The source of the second problem is the reliability of the control tables, kernel text, and drivers. A kernel crash means that some kind of inconsistency has occurred within the kernel and that there is a strong possibility a control structure has been damaged. As in-kernel crash dump mechanisms employ functions of the crashed system for outputting the dump, there is the very real possibility that the damaged control structures will be referenced. Besides, page tables and CPU registers such as the stack pointer may be corrupted too, which can potentially lead to faults during the crash dumping process. In these circumstances, even if a crash dump is finally obtained, the resulting dump image is likely to be invalid, so that it cannot be properly analyzed.

For in-kernel crash dumping mechanisms there is no obvious solution to the memory corruption problems. However, the locking issues can be alleviated by using polling mode (as opposed to interrupt mode) to communicate with the dump devices.

Setting up a controllable dump route within the kernel is very difficult, and this is increasingly true as the size and complexity of the kernel augments. This is what sparked the apparition of methods capable of capturing a dump independent from the existing kernel.

5.2 Kdump

Even though kdump proved to be much more reliable than in-kernel crash dumping mechanisms there are still issues in the three stages that constitute the dump process (see Section 2):

- Crash detection: hang detection, stack overflows, faults in the dump route.
- Minimal machine shutdown: stack overflows, faults in the dump route.
- Crash dump capture: device reinitialization, APIC reinitialization.

5.2.1 Stack overflows

In the event of a stack overflow critical data that usually resides at the bottom of the stack is likely to be stomped and, consequently, its use should be avoided.

In particular, in the i386 and IA64 architectures the macro `smp_processor_id()` ultimately makes use of the `cpu` member of struct `thread_info`, which resides at the bottom of the stack. x86_64, on the other hand, is not affected by this problem because it benefits from the use of the PDA infrastructure.

Kdump makes heavy use of `smp_processor_id()` in the reboot path to the second kernel, which can lead to unpredictable behaviour. This issue is particularly serious in SMP systems because not only the crashing CPU but also the rest of CPUs are highly dependent on likely-to-be-corrupted stacks. The reason it that during the minimal machine shutdown stage (see Section 2.2 for details) NMIs are used to stop the CPUs, but the NMI handler was designed on the premise that stacks can be trusted. This obviously does not hold good in the event of a crash overflow.

The NMI handler (see code listing 4) uses the stack indirectly through `nmi_enter()`, `smp_processor_id()`, `default_do_nmi`, `nmi_exit()`, and also through the crash-time NMI callback function (`crash_nmi_callback()`).

Even though the NMI callback function can be easily made stack overflow-safe the same does not apply to the rest of the code.

To circumvent some of these problems at the very least the following measures should be adopted:

- Create a stack overflow-safe replacement for `smp_processor_id`, which could be called `safe_smp_processor_id` (there is already an implementation for x86_64).

```
fastcall void do_nmi(struct
   pt_regs * regs, long error_code
   )
{
  int cpu;

  nmi_enter();
  cpu = smp_processor_id();
  ++nmi_count(cpu);

  if (!rcu_dereference(
     nmi_callback)(regs, cpu))
    default_do_nmi(regs);

  nmi_exit();
}
```

Listing 4: `do_nmi` (i386)

- Substitute `smp_processor_id` with `safe_smp_processor_id`, which is stack overflow-safe, in the reboot path to the second kernel.

- Add a new NMI low-level handling routine (`crash_nmi`) in `arch/*/kernel/entry.S` that invokes a stack overflow safe NMI handler (`do_crash_nmi`) instead of `do_nmi`.

- In the event of a system crash replace the default NMI trap vector so that the new `crash_nmi` is used.

If we want to be paranoid (and being paranoid is what crash dumping is all about after all), all the CPUs in the system should switch to new stacks as soon as a crash is detected. This introduces the following requirements:

- Per-CPU crash stacks: Reserve one stack per CPU for use in the event of a system crash. A CPU that has entered the dump route should switch to its respective per-CPU stack as soon as possible because the cause of the crash might have a stack overflow, and continuing to use the stack in such circumstances can lead to the generation of invalid faults (such as double fault or invalid TSS). If this happens the system is bound to either hang or reboot spontaneously. In SMP systems, the rest of the CPUs should follow suit, switching stacks at the NMI gate (`crash_nmi`).

- Private stacks for NMIs: The NMI watchdog can be used to detect hard lockups and invoke kdump. However, this dump route consumes a considerable amount of stack space, which could cause a stack overflow, or contribute to further bloating the stack if it has already overflowed. As a consequence of this, the processor could end up faulting inside the NMI handler which is something that should be avoided at any cost. Using private NMI stacks would minimize these risks.

To limit the havoc caused by bloated stacks, the fact that a stack is about to overflow should be detected before it spills out into whatever is adjacent to it. This can be achieved in two different ways:

- *Stack inspection*: Check the amount of free space in the stack every time a given event, such as an interrupt, occurs. This could be easily implemented using the kernel's stack overflow debugging infrastructure (`CONFIG_DEBUG_STACKOVERFLOW`).

- *Stack guarding*: The second approach is adding an unmapped page at the bottom of the stack so that stack overflows are detected at the very moment they occur. If a small impact in performance is considered acceptable this is the best solution.

5.2.2 Faults in the dump route

Critical parts of the kernel such as fault handlers should not make assumptions about the state of the stack. An example where proper checking is neglected can be observed in the code listing 5. The `mm` member of the struct `tsk` is dereferenced without making any checks on the validity of `current`. If `current` happens to be invalid, the seemingly inoffensive dereference can lead to recursive page faults, or, if things go really bad, to a triple fault and subsequent system reboot.

```
fastcall void __kprobes
   do_page_fault(struct pt_regs *
   regs, unsigned long error_code)
{
        struct task_struct *tsk;
        struct mm_struct *mm;

        tsk = current;
        .....
        [ no checks are made on
           tsk ]
        mm = tsk->mm;
        .....
}
```

Listing 5: `do_page_fault` (i386)

Finally, to avoid risks, control should be handed to kdump as soon as a crash is detected. The invocation of the dump mechanism should not be deferred to `panic` or `BUG`, because many things can go bad before we get there. For example, it is not guaranteed that the possible code paths never use any of the things that make assumptions about the current stack.

5.2.3 Hang detection

The current kernel has the necessary infrastructure to detect hard lockups and soft lockups, but they both have some issues:

- *Hard lockups*: This type of lockups are detected using the NMI watchdog, which periodically checks whether tasks are being scheduled (i.e. the scheduler is alive). The Achilles' heel of this detection method is that it is strongly vulnerable to stack overflows (see Section 5.2.1 for details). Besides, there is one inevitable flaw: hangs in the NMI handler cannot be detected.

- *Soft lockups*: There is a soft lockup detection mechanism implemented in the kernel that, when enabled (`CONFIG_DETECT_SOFTLOCKUP=y`), starts per-CPU watchdog threads which try to run once per second. A callback function in the timer interrupt handler checks the elapsed time since the watchdog thread was last scheduled, and if it exceeds 10 seconds it is considered a soft lockup.
  Currently, the soft lockup detection mechanism limits itself to just printing the current stack trace and a simple error message. But, the possibility of triggering the crash dumping process instead should be available.

Using LKDTT a case in which the existing mechanisms are not effective was discovered: hang with interrupts enabled (see *IRQ handling with IRQs enabled* in Section 4.1). In such scenario timer interrupts continue to be delivered and processed normally so both the NMI watchdog and the soft lockup detector end up judging that the system is running normally.

5.2.4 Device reinitialization

There are cases in which after the crash the capture kernel itself crashes attempting to initialize a hardware device.

In the event of a crash kdump does not do any kind of device shutdown and, what is more, the firmware stages of the standard boot process are also skipped. This may leave the devices in a state the second kernel cannot get them out of. The underlying problem is that the soft boot case is not handled by most drivers, which assume that only traditional boot methods are used (after all, many of the drivers were written before kexec even existed) and that all devices are in a reset state.

Sometimes even after a regular hardware reboot the devices are not reset properly. The culprit in such cases is a BIOS not doing its job properly.

To solve this issues the device driver model should be improved so that it contemplates the soft boot case, and kdump in particular. In some occasions it might be impossible to reinitialize a certain device without knowing its previous state. So it seems clear that, at least in some cases, some type information about the state of devices should be passed to the second kernel. This brings the power management subsystem to mind, and in fact studying how it works could be a good starting point to solve the device reinitialization problem.

In the meantime, to minimize risks each machine could have a dump device (a HD or NIC) set aside for crash dumping, so that the crash kernel would use that device and have no other devices configured.

5.2.5 APICs reinitialization

Kdump defers much of the job of actually saving the dump image to user-space. This means that kdump relies on the scheduler and, consequently, the timer interrupt to be able to capture a dump.

This dependency on the timer represents a problem, specially in i386 and x86_64 SMP systems. Currently, on these architectures, during the initialization of the kernel the legacy i8259 must exist and be setup correctly, even if it will not be used past this stage. This implies that, in APIC systems, before booting into the second kernel the interrupt mode must return to legacy. However, doing this is not as easy as it might seem because the location of the i8259 varies between chipsets and the ACPI MADT (Multiple APIC Description Table) does not provide this information. The return to legacy mode can accomplished in two different ways:

- *Save/restore BIOS APIC states*: All the APIC states are saved early in the boot process of the first kernel before the kernel attempts to initialize them, so that the APIC configuration as performed by the BIOS can be obtained. In the event of a crash, before booting into the capture kernel the BIOS APIC settings are restored back. Treating the APICs as black boxes like this has the benefit that the original states of the APICs can be restored even in systems with a broken BIOS. Besides, this method is theoretically immune to changes in the default configuration of APICs in new systems.
  There is one issue with this method though. It makes sure that the BIOS-designated boot CPU will always see timer interrupts in legacy mode, but this does not hold good if the second kernel boots on some other CPU as is possible with kdump. Therefore, for this method to work CPU relocation is necessary. It should also be noted that under certain rather unlikely circumstances relocation might fail (see Section 7.4 for details).

- *Partial save/restore*: Only the information that cannot be obtained any other way (i.e. i8259's location) is saved off at boot time. Upon a crash, taking into account this piece of information the APICs are reconfigured in such a way that all interrupts get redirected to the CPU in which the second kernel is going to be booted, which in kdump's case is the CPU where the crash occurred. This is the approach adopted by kdump.

6 LKDTT status and TODOS

Even though using LKDTT it is possible to test rather thoroughly the first two stages of the crash dumping process, that is *crash detection* and *minimal machine crash shutdown* (see Section 2), the capture kernel is not being sufficiently tested yet. The author is currently working on the following test cases:

- Pending IRQs: Leave the system with pending interrupts before booting into the capture kernel, so that the robustness of device drivers against interrupts coming at an unexpected time can be tested.

- Device reinitialization: For each device test whether it is correctly initialized after a soft boot.

Another area that is under development at the moment is test automation. However, due to the special nature of the functionality being tested there is a big roadblock for automation: the system does not always recover gracefully from crashes so that tests can resume. That is, in some occasions the crash dumping mechanism that is being tested will fail, or the system will hang while trying to reboot after capturing the dump. In such cases human intervention will always be needed.

7 Other kdump issues

The kernel community has been expecting that the various groups which are interested in crash dumping would converge around kdump once it was merged. And the same was expected from end-users and distributors. However, so far, this has not been the case and work has continued on other strategies.

The causes of this situation are diverse and, to a great extent, unrelated to reliability aspects. Instead, the main issues have to do with availability, applicability and usability. In some cases it is just a matter of time before they get naturally solved, but, in others, improvements need to be done to kdump.

7.1 Availability and applicability

Most of the people use distribution-provided kernels that are not shipped with kdump yet. Certainly, distributions will eventually catch up with the mainstream kernel and this problem will disappear.

But, in the mean time, there are users who would like to have a reliable crash dumping mechanism for their systems. This is especially the case of enterprise users, but they usually have the problem that updating or patching the kernel is not an option, because that would imply the loss of official support for their enterprise software (this includes DBMSs such as Oracle or DB2 and the kernel itself). It is an extreme case but some enterprise systems cannot even afford the luxury of a system reboot.

This situation along with the discontent with the crash dumping solutions provided by distributors sparked the apparition of other kexec-based projects (such as mkdump and Tough Dump), which were targeting not only mainstream adoption but also existing Linux distributions. This is why these solutions sometimes come in two flavors: a kernel patch for vanilla kernels and a fully modularized version for distribution kernels.

7.2 Usability

There are some limitations in kdump that have an strong impact in its usability, which affects both end-users and distributors as discussed below.

7.2.1 Hard-coding of reserved area's start address

To use kdump it is necessary to reserve a memory region big enough to accommodate the dump kernel. The start address and size of this region is indicated at boot time with the command line parameter `crashkernel=Y@X`, `Y` denoting how much memory to reserve, and `X` indicating at what physical address the reserved memory region starts. The value of `X` has to be used when configuring the capture kernel, so that it is linked to run from that start address. This means a displacement of the reserved area may render the dump kernel unusable. Besides it is not guaranteed that the memory region indicated at the command line is available to the kernel. For example, it could happen that the memory region does not exist, or that it overlaps system tables, such as ACPI's. All these issues make distribution of pre-compiled capture kernels cumbersome.

This undesirable dependency between the second and first kernel can be broken using a run-time relocatable kernel. The reason is that, by definition, a run-time relocatable kernel can run from any dedicated memory area the first kernel might reserve for it. To achieve run-time relocation a relocation table has to be added to the kernel binary, so that the actual relocation can be performed by either a loader (such as kexec) or even by the kernel itself. The first calls for making the kernel an ELF shared object. The second can be accomplished by resolving all the symbols in `arch/*/kernel/head.S` (this is what mkdump does).

7.2.2 Memory requirements

Leaving the task of writing out the crash dump to user space introduces great flexibility at the cost of increasing the size of the memory area that has to be reserved for the capture kernel. But for systems with memory restrictions (such as embedded devices) a really small kernel with just the necessary drivers and no user space may be more appropriate. This connects with the following point.

7.3 Kernel-space based crash dumping

After a crash the dump capture kernel might not be able to restore interrupts to a usable state, be it because the system has a broken BIOS, or be it because the interrupt controller is buggy. In such circumstances, processors may end up not receiving timer interrupts. Besides, the possibility of a timer failure should not be discarded either.

In any case, being deprived of timer interrupts is an insurmountable problem for user-space based crash dumping mechanisms such as kdump, because they depend on a working scheduler and hence the timer.

To tackle this problem a kernel-space driven crash dumping mechanism could be used, and even cohabit with the current user-space centered implementation. Which one to employ could be made configurable, or else, the kernel-space solution could be used as a fallback mechanism in case of failure to bring up user-space.

7.4 SMP dump capture kernel

In some architectures, such as i386 and x86_64, it is not possible to boot a SMP kernel from a CPU that is not the BIOS-designated boot CPU. Consequently, to do SMP in the capture kernel it is necessary to relocate to the boot CPU beforehand. Kexec achieves CPU relocation using scheduler facilities, but kdump cannot use the same approach because after a crash the scheduler cannot be trusted.

As a consequence, to make kdump SMP-capable a different solution is needed. In fact, there is a very simple method to relocate to the boot CPU that takes advantage of inter-processor NMIs. As discussed in Section 2.2 (*Minimal machine shutdown*), this type of NMIs are issued by the crashing CPU in SMP systems to stop the other CPUs before booting into the capture kernel. But this behavior can be modified so that relocation to the boot CPU is performed too. Obviously, if the crashing CPU is the boot CPU nothing needs to be done. Otherwise, upon receiving NMI the boot CPU should assume the task of capturing the kernel, so that the NMI-issuing CPU (i.e. the crashing the CPU) is relieved from that burden a can halt instead. This is the CPU relocation mechanism used by mkdump.

Even though being able to do SMP would boost the performance of the capture kernel, it was suggested that in some extreme cases of crash the boot CPU might not even respond to NMIs and, therefore, relocation to the boot CPU will not be possible. However, after digging through the manuals the author could only find (and reproduce using LKDTT) one such scenario, which occurs when the two conditions below are met:

- The boot CPU is already attending a different NMI (from the NMI watchdog for example) at the time the inter-processor NMI arrives.
- The boot CPU hangs inside the handler of this previous NMI, so it does not return.

The explanation is that during the time a CPU is servicing an NMI other NMIs are blocked, so a lockup in the NMI handler guarantees a system hang if relocation is attempted as described before. The possibility of such a hang seems remote and easy to evaluate. But it could also be seen as a trade-off between performance and reliability.

8 Conclusion

Existing testing methods for kernel crash dump capturing mechanisms are not adequate because they do not take into account the state of the hardware and the load conditions of the system. This makes it impossible to recreate many common crash scenarios, depriving test results of much of their validity. Solving these issues and providing controllable testing environment were the major motivations behind the creation of the LKDTT (Linux Kernel Dump Test Tool) testing project.

Even though LKDTT showed that kdump is more reliable than traditional in-kernel crash dumping solutions, the test results revealed some deficiencies in kdump too. Among these, minor hang detection deficiencies, great vulnerability to stack overflows, and problems reinitializing devices in the capture kernel stand out. Solutions to some of these problems have been sketched in this paper and are currently under development.

Since the foundation of the testing project the author could observe that new kernel releases (including release candidates) are sometimes accompanied by regressions. Regressions constitute a serious problem for both end-users and distributors, that requires regular testing and standardised test cases to be tackled properly. LKDTT aims at filling this gap.

Finally, several hurdles that are hampering the adoption of kdump were identified, the need for a run-time relocatable kernel probably being the most important of them.

All in all, it can be said that as far as kernel crash dumping is concerned Linux is heading in the right direction. Kdump is already very robust and most of the remaining issues are already being dealt with. In fact, it is just a matter of time before kdump becomes mature enough to focus on new fields of application.

9 Future lines of work

All the different crash dumping solutions do just that after a system crash: capture a crash dump. But there is more to a crashed system kexec than crash dumping. For example, in high availability environments it may be desirable to notify the backup system after a crash, so that the failover process can be initiated earlier.

In the future, kdump could also benefit from the current PID virtualization efforts, which will provide the foundation for process migration in Linux. The process migration concept could be extended to the crash case, in such a way that after doing some sanity-checking, tasks that have not been damaged can be migrated and resume execution in a different system.

Acknowledgements

I would like to express my gratitude to Itsuro Oda for his original contribution to LKDTT and valuable comments, as well as to all those who have laid the foundation for a reliable kernel crash dumping mechanism in Linux.

References

[1] Diskdump patches. `http://www.redhat.com/support/wpapers/redhat/netdump/`.

[2] Michael K. Johnson. Red Hat, Inc.'s network console and crash dump facility, 2002. `http://www.redhat.com/support/wpapers/redhat/netdump/`.

[3] Linux kernel crash dump (LKCD) home page, 2005. `http://lkcd.sourceforge.net/`.

[4] Hariprasad Nellitheertha. Reboot linux faster using kexec, 2004. `http://www-128.ibm.com/developerworks/linux/library/l-kexec.html`.

[5] Kexec-tools code. `http://www.xmission.com/~ebiederm/files/kexec/`.

[6] Vivek Goyal, Eric W. Biederman, and Hariprasad Nellitheertha. A kexec based dumping mechanism. In *Ottawa Linux Symposium (OLS 2005)*, July 2005.

[7] Kdump home page. `http://lse.sourceforge.net/kdump/`.

[8] Itsuro Oda. Mini Kernel Dump (MKDump) home page, 2006. `http://mkdump.sourceforge.net/`.

[9] Linux tough dump (TD) home page (japanese site), 2006. `http://www.hitachi.co.jp/Prod/comp/linux/products/solution.html`.

[10] Fernando Luis Vázquez Cao. Linux kernel dump test tool (LKDTT) home page, 2006. `http://lkdtt.sourceforge.net/`.

[11] EDAC wiki. `http://buttersideup.com/edacwiki/FrontPage`.

[12] Prasanna Panchamukhi. Kernel debugging with kprobes, 2004. `http://www-128.ibm.com/developerworks/linux/library/l-kprobes.html`.

[13] Djprobe documentation and patches. `http://lkst.sourceforge.net/djprobe.html`.

Exploring High Bandwidth Filesystems on Large Systems

Dave Chinner and Jeremy Higdon
Silicon Graphics, Inc.
`dgc@sgi.com` `jeremy@sgi.com`

Abstract

In this paper we present the results of an investigation conducted by SGI into streaming filesystem throughput on the Altix platform with a high bandwidth disk subsystem.

We start by describing some of the background that led to this project and our goals for the project. Next, we describe the benchmark methodology and hardware used in the project. We follow this up with a set of baseline results and observations using XFS on a patched 2.6.5 kernel from a major distribution.

We then present the results obtained from XFS, JFS, Reiser3, Ext2, and Ext3 on a recent 2.6 kernel. We discuss the common issues that we found to adversely affect throughput and reproducibility and suggest methods to avoid these problems in the future.

Finally, we discuss improvements and optimisations that we have made and present the final results we achieved using XFS. From these results we reflect on the original goals of the project, what we have learnt from the project and what the future might hold.

1 Background and Goals

In the past, there have been many comparisons of the different filesystems suppported by Linux. Most of these comparisons focus on activities typically performed by a kernel developer or use well known benchmark programs. Typically these tests are run on an average desktop machine with a single disk or, more rarely, a system with two or four CPUs with a RAID configuration of a few disks.

However, this really doesn't tell us anything about the maximum capabilities of the filesystems; these machine configurations don't push the boundaries of the filesystems and hence these observations have little relevance to those who are trying to use Linux in large configurations that require substantial amounts of I/O.

Over the past two years, we have seen a dramatic increase in the bandwidth customers require new machines to support. On older, modified 2.4.21 kernels, we could not achieve much more than 300 MiB/s on parallel buffered write loads. Now, on patched 2.6.5 kernels, customers are seeing higher than 1 GiB/s under the same loads. And, of course, there are customers who simply want all the I/O bandwidth we can provide.

The trend is unmistakable. A coarse correlation is that required I/O bandwidth matches the amount of memory in a large machine. Memory capacity is increasing faster than physical disk transfer rates are increasing, and this means that systems are being attached to larger numbers of disks in the hope that this provides higher throughput to populate and drain memory faster. Unfortunately, what we currently lack is any data on whether Linux can make use of the increased bandwidth that larger disk farms provide.

Some of the questions we need to answer include:

- How close to physical hardware limits can we push a filesystem?
- How stable is Linux under these loads?
- How does the Linux VM stand up to this sort of load?
- Do the Device Mapper (DM) and/or Multiple Device (MD) drivers limit performance or configurations?
- Are there NUMA issues we need to address?
- Do we have file fragmentation problems under these loads?
- How easily reproducible are the results we achieved and can we expect customers to be able to achieve them?
- What other bottlenecks limit the performance of a system?

To answer these questions, as they are important to SGI's customers, we put together a modestly sized machine to explore the limits of high-bandwidth I/O on Linux.

2 Test Hardware and Methodology

2.1 Hardware

The test machine was an Altix A3700 containing 24 Itanium2 CPUs running at 1.5 GHz in 12 nodes in a single cache-coherent NUMA domain. Each node is configured with 2 GiB of RAM for a system total of 24 GiB. Each node has 6.4 GB/s peak full duplex external interconnect bandwidth provided by SGI's NUMALink interconnect. A total of 12 I/O nodes, each with three 133 MHz PCI-X slots on two busses, were connected to the NUMALink fabric supplying 6.4 GB/s peak full duplex bandwidth per I/O node. The CPU and I/O nodes were connected via crossbar routers in a symmetric topology.

The I/O nodes were populated with a mix of U320 SCSI and Fibre Channel HBAs (64 SCSI controllers in total) and distributed 256 disks amongst the controllers in JBOD configuration. This provided an infrastructure that allowed each disk run at close to its maximum read or write bandwidth independently of any other disk in the machine.

The result is a machine with a disk subsystem theoretically capable of just over 11.5 GiB/s of throughput evenly distributed throughout the NUMALink fabric. Hence the hardware should be able to sustain maximum disk rates if the software is able to drive it that fast.

2.2 Methodology

The main focus of our investigation was on XFS performance. In particular, parallel sequential I/O patterns were of most interest as these are the most common patterns we see our customers using on their large machines. We also assessed how XFS compares with other mainstream filesystems on Linux on these workloads.

The main metrics we used to compare performance were aggregate disk throughput and CPU usage. We used multiple programs and independent test harnesses to validate the results against each other so we had confidence in the results of individual test runs that weren't replicated.

To be able to easily compare different configurations and kernels, we present normalised I/O efficiency results along with the aggregate throughput achieved. This gives an indication of the amount of CPU time being expended for each unit of throughput achieved. The unit of efficiency reported is `CPU/MiB/s`, or the percentage of a CPU consumed per mebibyte per second throughput. The lower the calculated number, the better the efficiency of the I/O executed.

The tests were run with file sizes large enough to make run times long enough to ensure that measurement was accurate to at least 1%. This, combined with running the tests in a consistent (scripted) manner, enabled us to draw conclusions about the reproducibility of the results obtained.

For most of the tests run, we used SGI's Performance Co-Pilot infrastructure [PCP] to capture high resolution archives of the system's behaviour during tests. This included disk utilisation and throughput, filesystem and volume manager behaviour, memory usage, CPU usage, and much more. We were able to analyse these archives after the fact which gave us great insight into system wide behaviour during the testing.

To find the best throughput under different conditions, we varied many parameters during testing. These included:

- different volume configurations
- the effect of I/O size on throughput and CPU usage
- buffered I/O and direct I/O
- different allocation methods for writes
- block device readahead
- filesystem block size
- pdflush tunables
- NUMA allocation methods

We tested several different kernels so we could chart improvements or regressions over time that our customers would see as they upgraded. Hence we tested XFS on SLES9 SP2, SLES9 SP3, and 2.6.15-rc5.

We also ran a subset of the above tests on other Linux filesystems including Ext2, Ext3, ReiserFS v3, and JFS. We kept as many configuration parameters as possible constant across these tests. Where supported, we used mkfs and mount parameters that were supposed to optimise data transfer rates and large filesystem performance.

The volume size for Ext2, Ext3, and ReiserFS V3 was halved to approximately 4.2 TiB because they don't support sizes of greater than 8 TiB. We took the outer portion of each disk for this smaller volume, hence maintaining the same stripe configuration. Compared to the larger volume used by XFS and JFS, the smaller volume has lower average seek times and higher minimum transfer rates and hence should be able to maintain higher average throughputs than the larger volume as the filesystems fill up during testing.

The comparison tests were scripted to:

1. Run mkfs with relevant large filesystem optimisations.
2. Make a read file set with `dd` by writing out the files to be read back with increasing levels of parallelism.
3. Perform buffered read tests using one file per thread across a range of I/O sizes and thread count measuring throughput, CPU usage, average process run time, and other metrics required for analysis.

The filesystem was unmounted and re-mounted between each test to ensure that all tests started without any cached filesystem data and memory approximately 99% empty.

4. Repeat Step 3 using buffered write tests, including truncating the file to be written in the overall test runtime.

Parallel writes were used to lay down the files for reading back to demonstrate the level of file fragmentation the filesystem suffered. The greater the fragmentation, the more seeking the disks will do and the lower the subsequent read rate achieved will be. Hence the read rate directly reflects on the fragmentation resistance of the filesystem. This is also a best case result because the tests are being run on an empty filesystem.

Finally, after we fixed several of the worst problems we uncovered, we re-ran various tests to determine the effect of the changes on the system.

2.3 Volume Layout and Constraints

Achieving maximum throughput from a single filesystem required a volume layout that enabled us to keep every disk busy at the same time. In other words, we needed to distribute the I/O as evenly as possible.

Building a wide stripe was the easiest way to achieve even distribution since we were mostly interested in sequential I/O performance. This exposed a configuration limitation of DM; `dmsetup` was limited to a line length of 1024 characters which meant we could only build a stripe approximately 90 disks wide.

Hence we ended up using a two level volume configuration where we had an MD stripe of 4 DM volumes each with 64 disks. We used an MD stripe of DM volumes because it was unclear whether DM and `dmsetup` supported multi-level volume configurations.

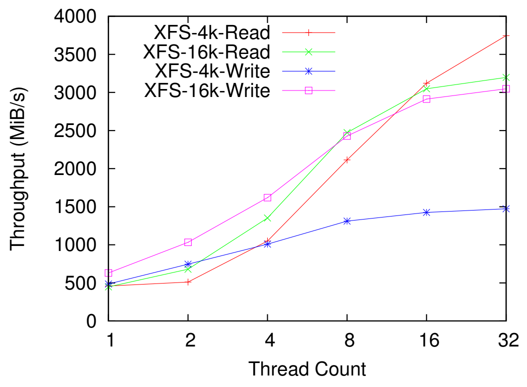


Figure 1: Baseline XFS Throughput

Using SGI's XVM volume manager, we were able to construct both a flat 256-disk stripe and a 4x64-disk multi-level stripe. Hence we were able to confirm that there was no measurable performance or disk utilisation difference between the two configurations.

Therefore we ran all the tests on the multi-level, MD-DM stripe volume layout. The only parameter that was varied in the layout was the stripe unit (and therefore stripe width) and most of the testing was done with stripe units of 512 KiB or 1 MiB.

3 Baseline XFS Results

Baseline XFS performance numbers were obtained from SuSE Linux Enterprise Server 9 Service Pack 2 (SLES9 SP2). We ran tests on XFS filesystem with both 4 KiB and 16 KiB block sizes. Performance varied little with I/O size, so the results presented used 128 KiB, which is in the middle of the test range.

Looking at read throughput, we can see from Figure 1 that there was very little difference between the different XFS filesystem configurations. In some cases the 16 KiB block size filesystem was faster, in other cases the 4 KiB block size filesystem was faster. Overall, they both averaged out at around 3.5 GiB/s across all block sizes.

In contrast, the 16 KiB block-size filesystem is substantially faster than the 4 KiB filesystem when writing. The 4 KiB filesystem appeared to be I/O bound as it was issuing much smaller I/Os than the 16KiB filesystem and the disks were seeking significantly more.

From the CPU efficiency graph in Figure 2, we can see that there is no difference in CPU time expended by the filesystem for different block sizes on read. This was expected from the throughput results.

Both the read and write tests show that CPU usage is scaling linearly with throughput; increasing the number of threads doing I/O does not decrease the efficiency of the filesystem. In other words, we are limited by either the rate at which we can issue I/Os or by something else outside the filesystem. Also, the write efficiency is substantially worse than for reads, it would seem that there is room for substantial improvement here.

4 Filesystem Comparison Results

The first thing to note about the results is that some of the filesystems were tested to higher numbers of threads and larger block sizes. The reasons for this were that some configurations were not stable enough to complete the whole test matrix and we had to truncate some of the longer test runs that would have prevented us from completing a full test cycle in our available time window. Consequently some of the results presented represent best-case performance rather than a mean of repeated test runs.

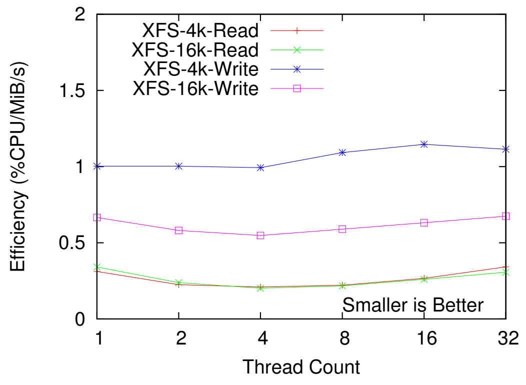


Figure 2: Baseline XFS Efficiency

The kernel used for all these tests was 2.6.15-rc5.

4.1 Buffered Read Results

The maximum read rates achieved by each filesystem can be seen in Figure 3. The read rate changed very little with varying I/O block size, we saw the same maximum throughput using 4 KiB I/Os as using 1 MiB I/Os. The only real difference was the amount of CPU consumed.

It is worth noting that XFS read throughput is substantially higher on 2.6.15-rc5 compared to the baseline results on SLES9 SP2. A discussion of this improvement canbe found in Section 6.2.

The performance of Ext2 and Ext3 is also quite different despite their common heritage. However, the results presented for Ext2 and Ext3 (as well as JFS) are the best of several test executions due to the extreme variability of the filesystem performance under these tests. The reasons for this variability are discussed in Section 5.2.

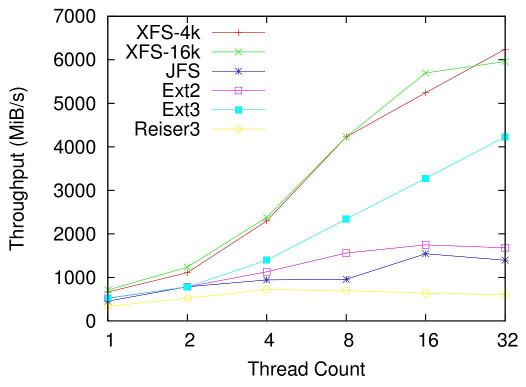


Figure 3: Buffered Read Throughput Comparison

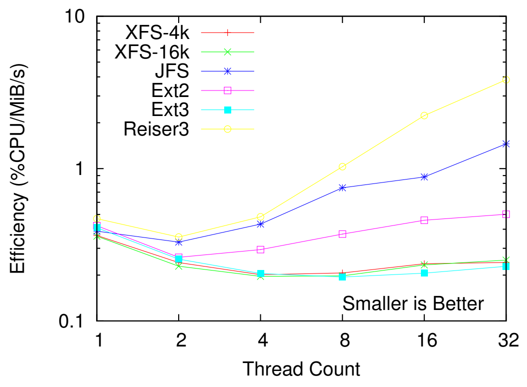


Figure 4: Buffered Read Efficiency Comparison

It is clear that XFS and Ext3 give substantially better throughput, and this is reflected in the efficiency plots in Figure 4, where these are the most efficient filesystems. Both ReiserFS and JFS show substantial decreases in efficiency as thread count increases. This behaviour is discussed in Section 5.1.

4.2 Buffered Write Results

Figure 5 shows some very clear trends in buffered write throughput. Firstly, XFS is substantially slower than the SLES9 SP2 baseline results. Secondly, throughput is peaking at four to eight concurrent writers for all filesystems except for Ext2. XFS, using a 16 KiB filesystem block size, was still faster than Ext2 until high thread counts were reached.

The poor write throughput of Ext3 and JFS is worth noting. JFS was unable to exceed an average of 80 MiB/s write speed in all but two of the many test points executed, and Ext3 did not score above 250 MiB/s and decreased to less than 100MiB/s at sixteen or more threads. We used the `data=writeback` mode for Ext3 as it was consistently 10% faster than the `data=ordered` mode.

The ReiserFS results are truncated due to problems running at higher thread counts. Writes would terminate without error unexpectedly, and sometimes the machine would hang. Due to time constraints this was not investigated further, but it is suspected that buffer initialisation problems which manifested on machines with both XFS and ReiserFS filesystems were the cause. The fixes did not reach the upstream kernel until well after testing had been completed [Scott][Mason].

JFS demonstrated low write throughput. We discovered that this was partially due to truncating a multi-gigabyte file taking several minutes to execute. However, the truncate time made up only half the elapsed time of each test. Hence, even if we disregarded the truncate time, JFS would still have had the lowest sustained write rate of all the filesystems.

Looking at the efficiency graph in Figure 6, we can see that only JFS and Ext2 had relatively flat profiles as the number of threads increased. However, the profile for JFS is relatively meaningless due to the low throughput. All the other filesystems show decreasing efficiency (increasing CPU time per MiB transferred to disk every second) at the same load points that they also showed decreasing

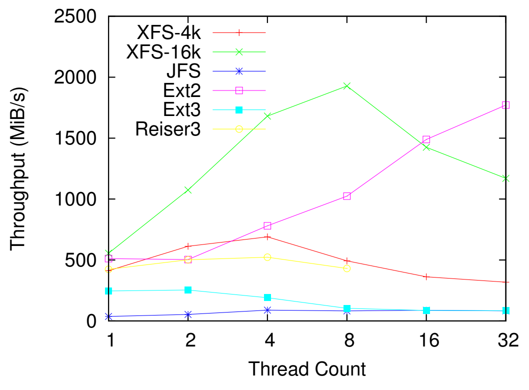


Figure 5: Buffered Write Throughput Comparison

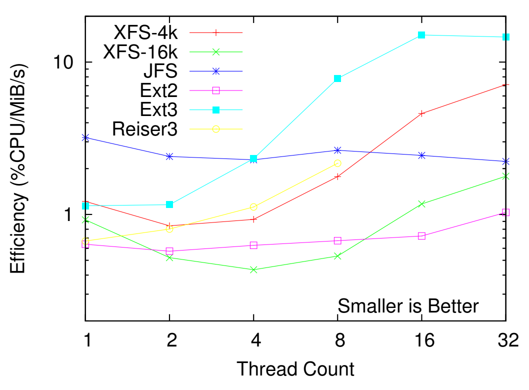


Figure 6: Buffered Write Efficiency Comparison

throughput. This is discussed further in Section 5.1.

4.3 Direct I/O Results

Only XFS and Ext3 were compared for direct I/O due to time constraints. The tests were run over different block sizes and thread counts, and involved first writing a file per thread, then overwriting the file, and finally reading the file back again. A 512 KiB stripe unit was used for these tests.

Table 1 documents the maximum sustained throughput we achieved with these tests. Ext3 was fastest with only a single thread, but writes still fell a long way behind XFS. As the number of threads increased, Ext3 got slower and slower as it fragmented the files it was writing. At 18 threads, Ext3 direct I/O performance was between 10 and 20 times lower than for a single thread.

Threads	FS	Read	Write	Overwrite
1	XFS	5.5	4.0	7.5
1	Ext3	4.2	0.6	2.5
18	XFS	10.0	7.7	7.7
18	Ext3	0.58	0.06	0.12

Table 1: Sequential Direct I/O Throughput (GiB/s)

In contrast, from one to 18 threads, XFS doubled its read and write throughput, and overwrite increased marginally from its already high single thread result. It is worth noting that the XFS numbers peaked substantially higher than the sustained throughput—reads peaked at above 10.7 GiB/s, while writes and overwrites peaked at over 8.9 GiB/s.

5 Issues Affecting Throughput

5.1 Spinlocks in Hot Paths

One thing that is clear from the buffered I/O results is that global spinlocks in hot paths of a filesystem do not scale. Every journalled filesystem except JFS was limited by spinlock contention during parallel writes. In the case of JFS, it appeared to be some kind of sleeping contention that limited performance, and so the impact of contention on CPU usage was not immediately measurable. Both ReiserFS and JFS displayed symptoms of contention in their read paths as well.

From analysis of the contention on the XFS buffered write path, we found that the contended lock was not actually being held for very long. The fundamental problem is the number of calls being made. For every page we write on a 4 KiB filesystem, we are allocating four filesystem blocks. We do this in four separate calls to `->prepare_write()`. Hence at the peak throughput of approximately 700 MiB/s, we are making roughly 180,000 calls per second that execute the critical section.

That gives us less than 5.6 microseconds to obtain the spinlock and execute our critical section to avoid contention. The code that XFS executes inside this critical section involves a function call, a memory read, two likely branches, a subtraction and a memory write. That is not a lot of code, but with enough CPUs trying to execute it in parallel it quickly becomes a bottleneck.

Of all the journalling filesystems, XFS appears to have the smallest global critical section in its write path. Filesystems that do allocation in the write path (instead of delaying it until later like XFS does) can't help but have larger critical sections here, and this shows in the throughput being achieved.

Looking to the future, we need to move away from allocating or mapping a block at a time in the generic write path to reduce the load on critical sections in the filesystems. While work is being done to reduce the number of block mapping calls on the read path, we need to do the same work for the write path. In the meantime, we have solved XFS's problem in a different way (see Section 6.1.2).

5.2 File Fragmentation and Reproducibility

From observation, the main obstacle in obtaining reproducible results across multiple test runs on each filesystem was file fragmentation. XFS was the only filesystem that almost completely avoided fragmentation of its working files. ReiserFS also seemed to be somewhat resistant to fragmentation but the results are not conclusive due to the problems ReiserFS had writing files in parallel.

Ext2, Ext3 and JFS did not resist fragmentation at all well. From truncated test results, we know that the variation was extreme. A comparison of the best case results versus the worst case results for ext2 can be seen in Table 2. Both Ext3 and JFS demonstrated very similar performance variation due to the different amounts of fragmentation of the files being read in each test run. While we present the best numbers we achieved for these filesystems, you should keep in mind that these are not consistently reproducible under real world conditions.

At the other end of the scale, the XFS results were consistently reproducible to within $\pm 3\%$. This is due to the fact that we rarely saw fragmentation on the XFS filesystems and the disk allocation for each file was almost identical on every test run. Even when we did see fragmentation, the contiguous chunks of file data were never smaller than several gigabytes in size.

A further measure of fragmentation we used was the number of physical disk I/Os required to provide the measured throughput. In the case of XFS, we were observing stripe unit sized I/Os being sent to each disk (512 KiB) while sustaining roughly 13,000 disk I/Os per second to achieve 6.3 GiB/s.

In contrast, Ext2 and Ext3 were issuing approximately 60–70,000 disk I/Os per second to achieve 1.7 GiB/s and 4.5 GiB/s respectively. That equates to average I/O sizes of approximately 24 KiB and 56 KiB and each disk executing more than 250 I/Os per second each. The disks were seek bound rather than bandwidth bound. Sustained read throughput of less

Threads	Best Run	Worst Run
1	522.2	348.5
2	780.2	74.8
4	1130.3	105.0
8	1542.1	176.8

Table 2: Example of Ext2 Read Throughput Variability (MiB/s)

than 300 MiB/s at 60–70,000 disk I/Os per second with an average size of 4 KiB was not uncommon to see. This indicates worst case (single block) fragmentation in the filesystem. The same behaviour was seen with JFS as well.

The source of the fragmentation on Ext2 and Ext3 would appear to be interleaved disk allocation when multiple files are written in parallel from multiple CPUs. This also occurred when running parallel direct I/O writes on Ext3 (see Table 1) so it would appear to be a general issue with the way Ext3 handles parallel allocation streams.

XFS solves this problem by decoupling disk block allocation from disk space accounting and then using well known algorithmic techniques to avoid lock contention to achieve write scaling.

The message being conveyed here is that most Linux filesystems do not resist fragmentation under parallel write loads. With parallelism hitting the mainstream now via multicore CPUs, we need to recognise that filesystems may not be as resistant to fragmentation under normal usage patterns as they were once recognised to be. This used to be a problem that only supercomputer vendors had to worry about. . .

5.3 kswapd and pdflush

While running single threaded tests, it was clear that there was something running in the background that was using more CPU time than the writer process and pdflush combined. A single threaded read from disk consuming a single CPU was consuming 10–15% of a CPU on each node running memory reclaim via kswapd. For a single threaded write, this was closer to 30% of a CPU per node. On our twelve node machine, this meant that we were using between 1.5 and 3.5 CPUs to reclaim memory being allocated by a single CPU.

PID	State	% CPU	Name
23589	R	97	dd
345	R	88	kswapd7
344	R	83	kswapd6
23556	R	81	dd
348	R	80	kswapd10
346	R	79	kswapd8
347	R	77	kswapd9
339	R	76	kswapd1
349	R	74	kswapd11
343	R	72	kswapd5
23517	R	71	dd
23573	R	64	dd
338	R	64	kswapd0
23552	R	64	dd
23502	R	63	dd
340	S	63	kswapd2
23570	R	61	dd
23592	R	60	dd
341	R	57	kswapd3

Table 3: kswapd CPU usage during buffered writes

On buffered write tests, pdflush also appeared to be struggling to write out the dirty data. With a single write thread, pdflush would consume very little CPU; maybe 10% of a single CPU every five seconds. As the number of threads increased, however, pdflush quickly became overwhelmed. At four threads writing at approximately 1.5 GiB/s, pdflush ran permanently consuming an entire CPU.

At eight or more write threads, pdflush consumed CPU time only sporadically; instead the kswapd CPU usage jumped from 30% of a CPU

Threads	Average I/O Size
1	1000 KiB
2	450 KiB
4	400 KiB
8	250 KiB
16	200 KiB
32	220 KiB

Table 4: I/O size during buffered writes

to 70–80% of a CPU per node. This can be seen in Table 3.

Monitoring of the disk level I/O patterns indicated that writeback was occuring from the LRU lists rather than in file offset order from pdflush. This could also be seen in the I/O sizes that were being issued to disk as seen in Table 4 as the thread count increased.

This is clearly not scalable writeback and memory reclaim behaviour; we need reclaim to consume less CPU time and for all writeback to occur in file offset order to maximise throughput. For XFS, this will also minimise fragmentation during block allocation. See Section 6.2.2 for details on how we improved this behaviour.

6 Improvements and Optimisations

6.1 XFS Modifications

6.1.1 Buffered Write I/O Path

In 2.6.15, a new buffered write I/O path implementation was introduced. This was written by Christoph Hellwig and Nathan Scott[Hellwig]. The main change this introduced was XFS clustering pages directly into a `bio` instead of by buffer heads and `submit_bh()` calls. Using buffer heads limited the size of an I/O to the number of buffer heads a `bio` could hold. In other words, the larger the block size of the filesystem, the larger the I/Os that could be formed in the write cluster path. This is the primary reason for the difference in throughput we see for the XFS filesystems with different block sizes.

By adding complete pages to a `bio` rather than buffer heads, we were able to make XFS write clustering independent of the filesystem block size. This means that any XFS filesystem can issue I/Os only limited in size by the number of pages that can be held by the `bio` vector.

Unfortunately, due to the locking issue described earlier in Section 5.1, XFS with the modified write path was actually slower on our test machine than without it. Clearly, the spinlock problem needed to be solved before we would see any benefit from the new I/O path.

6.1.2 Per-CPU Superblock Counters

Kernel profiles taken during parallel buffered write tests indicated contention within XFS on the in-core superblock lock. This lock protects the current in-core (in-memory) state of the filesystem.

In the case of delayed allocation, XFS uses the in-core superblock to track both disk space that is actually allocated on disk as well as the space that has not yet been allocated but is dirty in memory. That means during a `write(2)` system call we allocate the space needed for the data being written but we don't allocate disk blocks. Hence the "allocation" is very fast whilst maintaining an accurate representation of how much space there is remaining in the filesystem.

This makes contention on this structure a difficult problem to solve. We need global accuracy, but we now need to avoid global contention.

The in-core superblock is a write-mostly structure, so we can't use atomic operations or RCU to scale it. The only commonly used method remaining is to make the counters per-CPU, but we still need to have some method of being accurate when necessary that performs in an acceptable manner.

Hence for the free space counter we decided to trade off performance for accuracy when we are close to ENOSPC. The algorithm that was implemented is essentially a distributed counter that gets slower and more accurate as the aggregated total of the counter approaches zero.

When an individual per-CPU counter reaches zero, we execute a balance operation. This operation locks out all the per-CPU counters before aggregating and redistributing the aggregated value evenly over all the counters before re-enabling the counters again. This requires a per-CPU atomic exclusion mechanism. The balance operation must lock every CPU fast path out and so can be an expensive operation on a large machine.

However, on that same large machine, the fast path cost of the per-CPU counters is orders of magnitude lower than a global spinlock. Hence we are amortising the cost of an expensive rebalance very quickly compared to using a global spinlock on every operation. Also, when the filesystem has lots of free space we rarely see a rebalance operation as the distributed counters can sink hundreds of gigabytes of allocation on a single CPU before running dry.

If a counter rebalance results in a very small amount being distributed to each CPU, the counter is considered to be near zero and we fall back to a slow, global, single threaded counter for the aggregated total. That is, we prefer accuracy over blazing speed. It should also be noted that using a global lock in this case tends to be more efficient than constant rebalancing on large machines.

The results (see Figure 7 and Figure 8) speak for themselves and the code is to be released with 2.6.17[Chinner].

6.2 VM and NUMA Issues

6.2.1 SN2 Specific TLB Purging

When first running tests on 2.6.15-rc5, it was noticed that XFS buffered read speeds were much higher than we saw on SLES9 SP2, SLES9 SP3 and 2.6.14. On these kernels we were only achieving a maximum of 4 GiB/s. Using 2.6.15-rc5 we achieved 6.4 GiB/s, and monitoring showed all the disks at greater than 90% utilisation so we were now getting near to being disk bound.

Further study revealed that the memory reclaim rate limited XFS buffered read throughput. In this particular case, the global TLB flushing speed was found to make a large difference to the reclaim speed.

We found this when we reverted a platform-specific optimisation that was included in 2.6.15-rc1 to speed up TLB flushing[Roe]. Reverting this optimisation reduced buffered read throughput by approximately 30% on the same filesystem and files. Simply put, this improvement was an unexpected but welcome side effect of an optimisation made for different reasons.

6.2.2 Node Local Memory Reclaim

In a stroke of good fortune, Christoph Lameter completed a set of modifications to the memory reclaim subsystem[Lameter] while we were running tests. The modifications were included in Linux 2.6.16, and they modified the reclaim behaviour to reclaim clean pages on a given

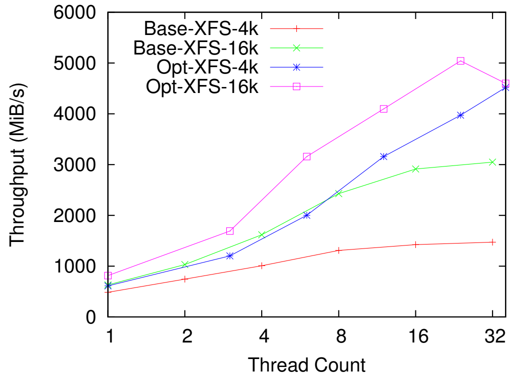


Figure 7: Improved XFS Buffered Write Throughput

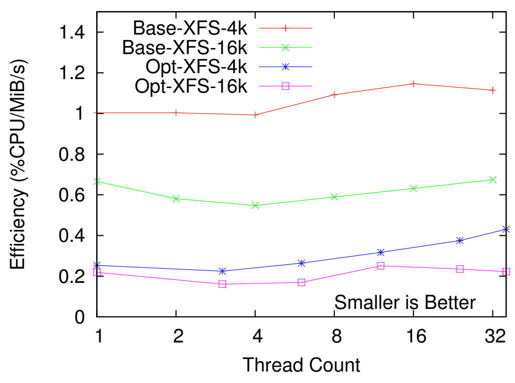


Figure 8: Improved XFS Buffered Write Efficiency

node before trying to allocate from a remote node.

The first major difference in behaviour was that kswapd never ran during either buffered read or write tests. Buffered reads were now quite obviously I/O bound with approximately half the disks showing 100% utilisation. Using a different volume layout with a 1 MiB stripe unit, sustained buffered read throughput increased to over 7.6 GiB/s.

The second most obvious thing was that pdflush was now able to flush more than 5 GiB/s of data whilst consuming less than half a CPU. Without the node local reclaim, it was only able to push approximately 500 MiB/s when it consumed an equivalent amount of CPU time. Writeback, especially at low thread counts, became far more efficient.

6.2.3 Memory Interleaving

While doing initial bandwidth characterisations using direct I/O, we found that it was necessary to ensure that buffer memory was allocated evenly from every node in the machine. This was achieved using the `numactl -i all` command prefix to the test commands being run.

Without memory interleaving, direct I/O (read or write) struggled to achieve much more than 6 GiB/s due to the allocation patterns limiting the buffers to only a few nodes in the machine. Hence we were limited by the per-node NUMALink bandwidth. Interleaving the buffer memory across all the nodes solved this problem.

With buffered I/O, however, we saw very different behaviours. In initial testing we saw little difference in throughput because the page cache ended up spread across all nodes of the machine due to memory reclaim behaviour.

However, when testing the node local memory reclaim patches we found that interleaving did make a big difference to performance as the local reclaim reduced the number of nodes that the page cache ended up spread over. Interestingly, the improvement in memory reclaim speed that the local reclaim gave us meant that there was no performance degradation despite not spreading the pages all over the machine. Once we spread the pages using the `numactl` command we saw the substantial performance increases.

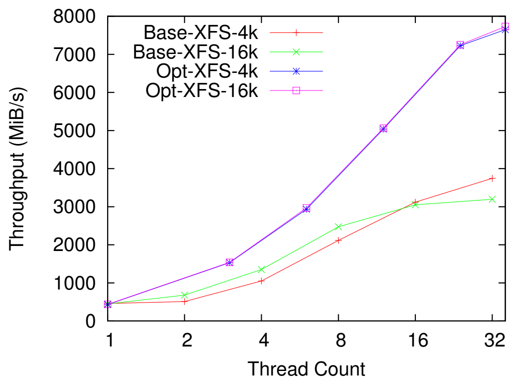


Figure 9: Improved XFS Buffered Read Throughput

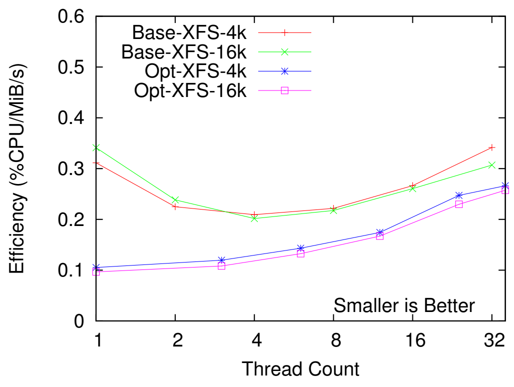


Figure 10: Improved XFS Buffered Read Efficiency

6.2.4 Results

We've compared the baseline buffered I/O results from Section 3 with the best results we achieved with our optimised kernel.

From Figure 7 it is clear that we achieved a substantial gain in write throughput. The outstanding result is the improvement of 4 KiB block size filesystems and is a direct result of the I/O path rewrite. The improved write clustering resulted in consistently larger I/Os being sent to disk, and this has translated into improved throughput. Local memory reclaim has also prevented I/O sizes from decreasing as the number of threads writing increases which has also contributed to higher throughputs as well.

On top of improved throughput, Figure 8 indicates that the buffered write efficiency has improved by factor of between three and four. It can been seen that the efficiency decreases somewhat as throughput and thread count goes up, so there is still room for improvement here.

Buffered read throughput has roughly doubled as shown in Figure 9. This improvement can be almost entirely attributed to the VM improvements as the XFS read path is almost identical in the baseline and optimised kernels.

Once again, the improvement in throughput corresponds directly to an improvement in efficiency. Figure 10 indicates that we saw much greater improvements in efficiency at low thread counts than at high thread counts. The source of this decrease in efficiency is unknown and more investigation is required to understand it.

One potential reason for the decrease in efficiency of the buffered read test as throughput increases is that the NUMALink interfaces may be getting close to saturation. With the tests being run, the typical memory access patterns are a DMA write from the HBA to memory, which due to the interleaved nature of the page cache is distributed across the NUMALink fabric. The data is then read by a CPU, which gathers the data spread across every node, and is then written back out into a user buffer which is spread across every node.

With both bulk data and control logic memory references included, each node node in the system is receiving at least 2 GiB/s and transmitting more than 1.2 GiB/s. With per-node receive throughput this high, remote memory read and write latencies can increase compared to an idle interconnect. Hence the increase in CPU usage may simply be an artifact of sustained high NUMALink utilisation.

7 Futures

The investigation that we undertook has provided us with enough information about the behaviour of these large systems for us to predict issues that SGI customers will see over the next year or two. It has also demonstrated that there are issues that mainstream Linux users are likely to start to see over this same timeframe. With technologies like SAS, PCI express, multicore CPUs and NUMA moving into the mainstream, issues that used to affect only high end machines are rapidly moving down to the average user. We need to make sure that our filesystems behave well on the average machine of the day.

At the high end, while we are on top of filesystem scaling issues with XFS, we are starting to see interactions between high bandwidth I/O and independent cpuset constrained jobs on large machines. These interactions are complex and are hinting that for effective deployment on large machines at high I/O bandwidths the filesystem needs to be NUMA and I/O path topology aware so that filesystem placement and I/O bandwidth locality to the running job can be maximised. That is, we need to be able to control placement in the filesystem to minimise the NUMALink bandwidth that a job's I/O uses.

This means that filesystems are likely to need allocation hints provided to them to enable this sort of functionality. We already have policy information controlling how a job uses CPU and memory in large machines, so extending this concept to how the filesystem does allocation is not as far-fetched as it seems.

Improving performance in filesystems is all about minimising disk seeking, and this comes down to the way the filesystem allocates its disk space. We have new issues at the high end to deal with, while the issues that have been solved at the high end are now becoming issues for mainstream. As the intrinsic parallelism of the average computer increases, algorithms need to be able to resist fragmentation when allocations occur simultaneously so that filesystem performance can grow with machine capability.

8 Conclusion

The investigation we undertook has provided us with valuable information on the behaviour of Linux in high bandwidth I/O loads. We identified several areas which limited our performance and scalability and fixed the worst during the investigation.

We improved the efficiency of buffered I/O under these loads and significantly increased the throughput we could achieve from XFS. We discovered interesting NUMA scalability issues and either fixed them or developed effective strategies to negate the issues.

We proved that we could achieve close to the physical throughput limits of the disk subsystem with direct I/O. From analysis, we found that even buffered I/O was approaching physical NUMALink bandwidth limits. We proved that Linux and XFS in combination could do this whilst maintaining reproducible and stable operation.

We also uncovered a set of generic filesystem issues that affected every filesystem we tested. We solved these problems on XFS, and provided recommendations on why we think they also need to be solved.

Finally, we proved that XFS is the best choice for our customers; both on the machines they use and for the common workloads they run.

In conclusion, our investigation fulfilled all the goals we set at the beginning of the task. We gained insight into future issues we are likely to see, and we raised a new set of questions that need further research. Now all we need is a bigger machine and more disks.

References

[PCP] Silicon Graphics Inc., *Performance Co-Pilot*, `http://oss.sgi.com/projects/pcp/`

[Scott] Nathan Scott, *Make alloc_page_buffers() initialise buffer_heads using init_buffer()*. Git commit key: `01ffe339e3a0ba5ecbeb2b3b5abac7b3ef90f374`

[Mason] Chris Mason, *[PATCH] reiserfs: zero b_private when allocating buffer heads*. Git commit key: `fc5cd582e9c934ddaf6f310179488932cd154794`

[Roe] Dean Roe, *[IA64] - Avoid slow TLB purges on SGI Altix systems*. Git commit key: `c1902aae322952f8726469a6657df7b9d5c794fe`

[Lameter] Christoph Lameter, *[PATCH] Zone reclaim: Reclaim logic*. Git commit key: `9eeff2395e3cfd05c9b2e6074ff943a34b0c5c21`

[Hellwig] Christoph Hellwig and Nathan Scott, *[XFS] Initial pass at going directly-to-bio on the buffered IO path*. Git commit key: `f6d6d4fcd180f8e47bf6b13fc6cce1e6c156d0ea`

[Chinner] Dave Chinner, *[XFS] On machines with more than 8 cpus, when running parallel I/O*. Git commit key: `8d280b98cfe3c0b69c37d355218975c1c0279bb0`

The Effects of Filesystem Fragmentation

Giel de Nijs
Philips Research
giel.de.nijs@philips.com

Ard Biesheuvel
Philips Research
ard.biesheuvel@philips.com

Ad Denissen
Philips Research
ad.denissen@philips.com

Niek Lambert
Philips Research
niek.lambert@philips.com

Abstract

To measure the actual effects of the fragmentation level of a filesystem, we simulate heavy usage over a longer period of time on a constantly nearly full filesystem. We compare various Linux filesystems with respect to the level of fragmentation, and the effects thereof on the data throughput. Our simulated load is comparable to prolonged use of a Personal Video Recorder (PVR) application.

1 Introduction

For the correct reading and writing of files stored on a block device (*i.e.*, hard drive, optical disc, etc.) the actual location of the file on the platters of the disk is of little importance; the job of the filesystem is to transparently present files to higher layers in the operating system. For *efficient* reading and writing, however, the actual location does matter. As blocks of a file typically need to be presented in sequence, they are mostly also read in sequence. If the blocks of a file are scattered all over a hard drive, the head of the drive needs to seek to subsequent blocks very often, instead of just reading those blocks in one go. This takes time and energy, and so the effective transfer speed of the hard drive is lower and the energy spent per bit read is higher. Obviously, one wants to avoid this.

In the early days of the PC, the filesystem of choice for many was Microsoft's FAT [1], later followed by FAT32. The allocation strategy, the strategy that determines which blocks of a file go where on the disk, was very simple: write every block of the file to the first free block found. On an empty hard drive, the blocks will be contiguous on the disk and reading will not involve many seeks. As the filesystem ages and files are created and deleted, the free blocks will be scattered over the drive, as will newly created files. The files on the hard drive become fragmented and this affects the overall drive performance. To solve this, a process called *defragmentation* can re-order blocks on the drive to ensure the blocks of each file and of the remaining free space will be contiguous on the disk. This is a time consuming activity, during which the system is heavily loaded.

More sophisticated filesystems like Linux's ext2 [2] incorporate smarter allocation strategies, which eliminate the need for defragmentation for everyday use. Specific usage scenarios might exist where these, and other, filesystems perform significantly worse or better than average. We have explored such a scenario and derive a theoretical background that can predict, up to a point, the stabilisation of the fragmentation level of a filesystem.

In this paper we study the level of fragmentation of various filesystems throughout their lifetime, dealing with a specific usage scenario. This scenario is described in section 2 and allows us to derive formulae for the theoretical fragmentation level in section 3. We elaborate on our simulation set-up in section 4, followed by our results in section 5. We end with some thoughts on future work in section 6 and conclude in section 7.

2 The scenario

Off-the-shelf computers are used more and more for storing, retrieving and processing very large video files as they assume the role of a more advanced and digital version of the classic VCR. These so-called Personal Video Recorders (PVR) are handling all the television needs of the home by recording broadcasts and playing them back at a time convenient for the user, either on the device itself or by streaming the video over a home network to a separate rendering device. Add multiple tuners and an ever increasing offer of broadcast programs to the mix and you have an application that demands an I/O subsystem that is able to handle the simultaneous reading and writing of a number of fairly high bandwidth streams. Both home-built systems as well as Consumer Electronics (CE) grade products with this functionality exist, running software like MythTV [3] or Microsoft Windows Media Center [4] on top of a standard operating system.

As these systems are meant to be always running, power consumption of the various components becomes an issue. The costs of the components is of course an important factor as well, especially for CE devices. Furthermore, the performance of the system should not deteriorate over time to such a level that it becomes unusable, as a PVR should be low maintenance and should just work. Clearly, overdimensioning the system to overcome performance issues is not the preferred solution. A better way would be to design the subsystems in such a way that they are able to deliver the required performance efficiently and predictably. As stated above, the hard-disk drive (HDD) will be one of the most stressed components, so it is interesting to see if current solutions are fulfilling our demands.

2.1 Usage pattern

The task of a PVR is mainly to automatically record broadcast television programs, based on a personal preference, manual instruction or a recommendation system. As most popular shows are broadcast around the same time of day and PVRs are often equipped with more than one tuner, it is not uncommon that more than one program is being recorded simultaneously. As digital broadcast quality results in video streams of about 4 to 8 megabit/s, the size of a typical recording is in the range of 500 MB to 5 GB.

As the hard drive of a PVR fills up, older recordings are deleted to make room for newer ones. The decision which recording to delete is based on age and popularity, *e.g.*, the news of last week can safely be deleted, but a user might want to keep a good movie for a longer period of time.

The result of this is that the system might be writing two 8 megabit/s streams to a nearly full filesystem, sometimes even while playing back one or more streams. For 5 to 10 recordings per day, totalling 3 to 5 hours of content, this results in about 10 GB of video data written to the disk. Will the filesystem hold up if this is done daily for two years? Will the average amount of fragmentation keep increasing or will it stabilise at some point? Will the effective data rate when reading the recorded files from a fresh filesystem differ from the data rate when reading from one that has been used extensively? We hope to answer these questions with our experiments.

Although the described scenario is fairly specific, it is one that is expected to be increasingly important. The usage and size of media files are both steadily increasing and general Personal Computer (PC) hardware is finding its way into CE devices, for which cost and stability are main issues. The characterised usage pattern is a general filesystem stress test for situations involving large media files.

As an interesting side-note, our scenario describes a pattern that had hardly ever been encountered before. Normal usage of a computer system slowly fills the hard drive while reading, writing and deleting. If the hard drive is full, it is often replaced by a bigger one or used for read-only storage. Our PVR scenario, however, describes a full filesystem that remains in use for reading and writing large files over a prolonged period of time.

2.2 Performance vs. power

A filesystem would ideally be able to perform equally well during the lifetime of the system it is part of, without significant performance loss due to fragmentation of the files. This is not only useful for shorter processing times of non-real-time tasks (*e.g.*, the detection of commercial blocks in a recorded television broadcast), but it also influences the power consumption of the system [5].

If a real-time task with a predefined streaming I/O behaviour is running, such as the recording of a television program, power can be saved if the average bit rate of the stream is lower than the maximum throughput of the hard drive. If a memory buffer is assigned for low power purposes, it can be filled as fast as possible by reading the stream from the hard drive and powering down the drive while serving the application from the memory buffer. This also holds for writing: the application can write into the memory buffer, which can be flushed to disk when it is full, allowing us to power off the drive between bursts. The higher the effective read or write data rate is, the more effective this approach will be. If the fragmentation of the filesystem is such that it influences the effective data rate, it directly influences the power consumption. A system that provides buffering capabilities for streaming I/O while providing latency guarantees is ABISS [6].

3 The theory

To derive a theory dealing with the fragmentation level of a filesystem, we first need some background information on filesystem allocation. This allocation can (and will) lead to fragmentation, as we will describe below. We determine what level of fragmentation is acceptable and as a result we can derive the fragmentation equilibrium formulae of section 3.3.

3.1 Block allocation in filesystems

Each filesystem has some sort of rationale that governs which of the available blocks it will use next when more space needs to be allocated. This rationale is what we call the allocation strategy of a filesystem. Some allocation strategies are more sophisticated than others. Also, the allocation strategy of a particular filesystem can differ between implementations without sacrificing interoperability, provided that every implementation meets the specifications of how the data structures are represented on disk.

3.1.1 The FAT filesystem

The *F*ile *A*llocation *T*able *F*ile *S*ystem (FATFS) [1] is a filesystem developed by Microsoft in the late seventies for its MS-DOS operating system. It is still in wide use today, mainly for USB flash drives, portable music players and digital cameras. While recent versions of Windows still support FATFS, it is no longer the default filesystem on this platform.

A FATFS volume consists of a boot sector, two file allocation tables (FATs), a root directory and a collection of files and subdirectories spread out across the disk. Each entry of the FAT maps to a cluster in the data space of the disk, and contains the index number of the next cluster of the file (or subdirectory) it belongs to. An index of zero in the FAT means the corresponding cluster on the disk is free, other magic numbers exist that denote that a cluster is the last cluster of a file or that the cluster is damaged or reserved. The second FAT is a backup copy of the first one, in case the first one gets corrupted.

An instance of FATFS is characterised by three parameters: the number of disk sectors in a cluster (2^i for $0 \leq i \leq 7$), the number of clusters on the volume and the number of bits used for each FAT entry (12, 16 or 32 bits), which at least equals the base-2 logarithm of the number of clusters.

The allocation strategy employed by FATFS is fairly straight-forward. It scans the FAT linearly, and uses the first free cluster found. Each scan starts from the position in the FAT where the previous scan ended, which results in all of the disk being used eventually, even if the filesystem is never full. However, this strategy turns out to be too naive for our purpose: if several files are allocated concurrently, the files end up interleaved on the disk, resulting in high fragmentation levels even on a near-empty filesystem.

3.1.2 The LIMEFS filesystem

The *L*arge-file metadata-*I*n-*M*emory *E*xtent-based *F*ile *S*ystem (LIMEFS) was developed as a research venture within Philips Research. [7]

LIMEFS is extent-based, which means it keeps track of used and free blocks in the filesystem by maintaining lists of (*index*, *count*) pairs. Each pair is called an *extent*, and describes a set of *count* contiguous blocks starting at position *index* on the disk.

The allocation strategy LIMEFS uses is slightly more sophisticated than the strategy FAT incorporates, and turns out to be very effective in avoiding fragmentation when dealing with large files. When space needs to be allocated, LIMEFS first tries to allocate blocks in the free extent after the last written block of the current file. If there is no such extent, the list of free extents is scanned and an extent is chosen that is not preceded by an extent that contains the last block of another file that is currently open for writing. If it finds such an extent, it will start allocating blocks from the beginning of the extent. If it cannot find such an extent, it will pick an extent that *is* preceded by a file that is open for writing. In this case however, it will split the free extent in half and will only allocate from the second half. When the selected extent runs out of free blocks, another extent is selected using the approach just described, with the added notion that extents close to the original one are preferred over more distant ones.

3.2 How to count fragments

A hard drive reading data sequentially is able to transfer, on average, the amount of data present on one track in the time it takes the platter to rotate once. Reading more data than one track involves moving the head to the next track, which is time lost for reading. The layout of the data on the tracks is corrected for the track-to-track seek time of the hard drive, *i.e.*, the data is laid out in such a way that the first block on the next track is just passing underneath the head the moment it has moved from the previous track and is ready for reading. This way no additional time is left waiting for the right block. This is known as *track skew*, shown in figure 1. Hence, the effective transfer rate for sequential reading is the amount of data present on one track, divided by the rotation time increased with the track skew.

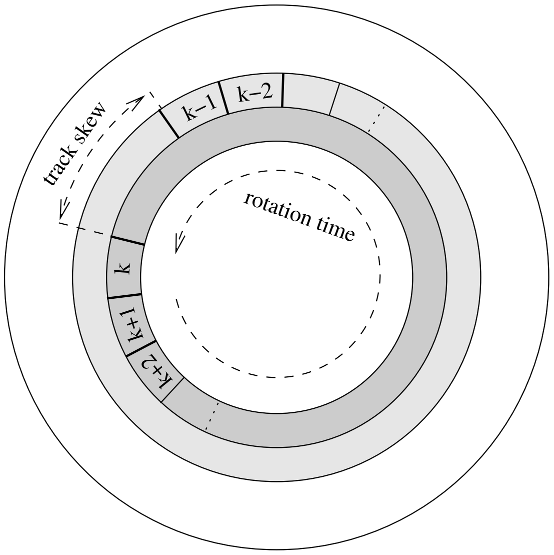


Figure 1: In the time it takes the head of the hard drive to move to the next track, the drive continues to rotate. The lay-out of blocks compensates for this. After reading block k, the head moves to the next track and arrives just in time to continue reading from block $k+1$ onwards. This is called *track skew*.

When a request is issued for a block that is not on the current track, the head has to *seek* to the correct track. Subsequently, after the seek the drive has to wait until the correct block passes underneath the head, which takes on average half the rotation time of the disk. The time such a seek takes is time not spent reading, thus seeking lowers the data throughput of the drive. Seeks occur for example when blocks of a single non-contiguous file are read in sequence, *i.e.*, when moving from fragment to fragment. Some non-sequential blocks do not induce seeks however, and should not be counted as fragments.

Two consecutive blocks in a file that are not contiguously placed on the disk only lead to a seek if the seek time is lower than the time it would take to bridge the gap between the two blocks by just waiting for the block to turn up beneath the head (maybe after a track-to-track seek).

If t_s is the average time to do a full seek, and t_r is the rotation time of the disk, we can derive T_s, the access time resulting from a seek:

$$T_s = t_s + \frac{1}{2}t_r \qquad (1)$$

The access time resulting from waiting, T_w, can be expressed in terms of track size s_t, gap size s_g and track skew t_{ts}:

$$T_w = \frac{s_g}{s_t}(t_r + t_{ts}) \qquad (2)$$

So the maximum gap size S_g that does not induce a seek is the largest gap size s_g for which $T_w \leq T_s$ still holds, and therefore

$$S_g = s_t \frac{\frac{t_s}{t_r} + \frac{1}{2}}{1 + \frac{t_{ts}}{t_r}} \qquad (3)$$

The relevance of the maximum gap size S_g is that it allows us to determine how many rotations and how many seeks it takes to read a particular file, given the layout of its blocks on the disk.

3.3 Fragmentation equilibrium

A small amount of fragmentation is not bad per se, if it is small enough to not significantly reduce the transfer speed of a hard drive. For instance, the ext3 filesystem [8] by default inserts one metadata block on every 1024 data blocks. Although, strictly speaking, this leads to non-sequential data and thus fragmentation, this kind of fragmentation does not impact the data rate significantly. If anything, it increases performance because the relevant metadata is always nearby.

A more important factor is the predictability and the stabilisation of the fragmentation level. Dimensioning the I/O subsystem for a certain application is only possible if the effective data transfer rate of the hard drive is known a priori, *i.e.*, predictable. As one should dimension a system for the worst case, it is also helpful if the fragmentation level has an upper bound, *i.e.*, it reaches a maximum at some point in time after which it does not deteriorate any further. With the help of some simple mathematics, we can estimate the theoretical prerequisites for such stabilisation.

3.3.1 Neighbouring blocks

Suppose we have a disk with a size of N allocation blocks and from these blocks, we make a random selection of M blocks. The first selected block will of course never be a neighbour of any previously selected blocks. The probability that the second block is a neighbour of the first is $\frac{2}{N-1}$, because 2 of the remaining $N-1$ blocks are adjacent to the first block. The probability of the third block being a neighbour of one of the previous two is then $\frac{4}{N-2}$, or, more general, the probability of a block i being a neighbour of one of the previously selected blocks is:

$$P(i) = \frac{2i}{N-i} \qquad (1 \le i \ll N) \tag{4}$$

The average number of neighbouring blocks when randomly selecting M blocks, the expected value, can be determined by the summation of the probability of a selected block being a neighbour of a previously selected block over all blocks, although with two errors: we are not correcting for blocks at the beginning or end of the disk with only one neighbour and we are conveniently forgetting that two previously selected blocks could already be neighbours. As long as $N \gg 1$ and $M \ll N$ this will not influence the outcome very much and simplifies the formulae. Furthermore, if $M \ll N$ we can approximate (4) by:

$$P(i) \approx \frac{2i}{N} \tag{5}$$

The expected value of the number of neighbouring blocks is then, according to (5):

$$\begin{aligned} E &= \sum_{i=1}^{M-1} \frac{2i}{N-i} \\ &\approx \sum_{i=1}^{M-1} \frac{2i}{N} \\ &= \frac{(M-1)M}{N} \end{aligned} \tag{6}$$

3.3.2 Neighbouring fragments

The above holds for randomly allocated blocks, which is something not many filesystems do. The blocks in the above argumentation however can also be seen as fragments of both files as well as free space. This only changes the meaning of N and M and does not change the argumentation. If we now look at a moderately fragmented drive, the location of the fragments will be more or less random and our estimation of the expected value of the number of neighbouring fragments will be closer to the truth.

Furthermore, to have a stable amount of fragments, an equilibrium should exist between the number of new fragments created during allocation and the number of fragments eliminated during deletion (by deleting data next to a free fragment and thus 'glueing' two fragments).

Let us interpret N as the total number of fragments on a drive, both occupied as well as free and M as the number of free fragments. Also, assume that we are dealing with a drive that is already in use for some time and is almost full, according to our PVR scenario as described in section 2.1.

The fore mentioned equilibrium can be obtained with the simple allocation strategy of LIMEFS (described in section 3.1.2) used in our PVR scenario, by eliminating one fragment when deleting a file, as on average one fragment is created during the allocation of a file (as shown in figure 2). We assume that, as the files are on average of the same size, for each file created a file is deleted; when writing a file of L fragments (increasing the fragment count with one), a file of L fragments is deleted. This should eliminate one fragment to create a balance in the amount of fragments.

By deleting a file of L fragments on a drive with N total fragments, we increase the number of

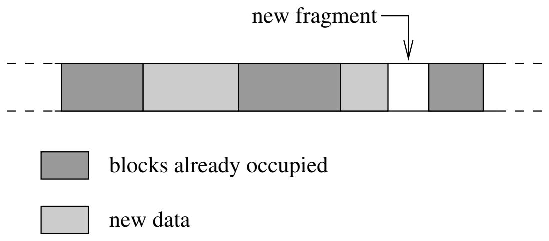


Figure 2: Empty fragments are filled (and thus not creating more fragmentation), until the end of the file. As an empty fragment is now divided into a fragment of the new file and a smaller empty fragment, one new fragment is created.

free fragments from $(M-L)$ to M. The number of neighbouring fragments n before deleting and n_d after deleting the file amongst the free fragments is, according to (6):

$$n = \frac{(M-L-1)(M-L)}{N} \tag{7}$$

$$n_d = \frac{(M-1)M}{N} \tag{8}$$

Combining (7) and (8) gets us the increase of neighbouring free fragments and thus the decrease of free fragments f (two neighbouring free fragments are one fragment):

$$\begin{aligned} f &= n_d - n \\ &= \frac{(M-1)M}{N} - \frac{(M-L-1)(M-L)}{N} \\ &= \frac{(2M-L-1)L}{N} \end{aligned} \tag{9}$$

3.3.3 Balancing fragmentation

Now we define two practical, understandable parameters m for the fraction of the disk that is free and s for the fraction of the disk occupied by a file:

$$m = \frac{M}{N} \tag{10}$$

$$s = \frac{L}{N} \tag{11}$$

This is mathematically not correct, as we defined N, M and L as distributions of an amount of fragments, but in the experiments of section 4 we will see that it works well enough for us, together with the numerous assumptions we already made.

To fulfil the demand that the number of eliminated fragments f when deleting a file of L fragments should equal the number of created fragments by allocating space for a file of an average equal size, $f = 1$ in (9). We combine this statement with (10) and (11):

$$L = \frac{1+s}{2m-s} \tag{12}$$

This can be simplified even more by assuming $s \ll 1$, which is true if $L \ll N$. This is a realistic assumption, as the size of a file is most often a lot smaller the the size of the disk. We get:

$$L = \frac{1}{2m-s} \tag{13}$$

A remarkable result of this equation is that the average number of fragments in each of our files does not depend in the allocation unit size. Of course, the above only holds if files consist of more than one fragment, and a fragment consists of a fairly large number of allocation units. This fits our PVR scenario, but the deduced formulae do not apply to general workloads.

Another interesting result of (13) is the insight that a disk without free space besides the amount needed to write the next file will result in a situation where $M = L$ and thus $m = s$. The average number of fragments in a file becomes $L = \frac{1}{s}$, meaning the number of fragments in a file equals the number of files on the disk.

4 The simulation

To test the validity of our theory described in section 3 on the one hand and to be able to assess if fragmentation is an issue for PVR systems on the other hand, we have conducted a number of simulations. First, we have done in-memory experiments with the simple allocation strategy of LIMEFS [7]. Next, we have tried simulating a real PVR system working on a filesystem as closely as possible with our program `pvrsim`. This was combined with our `hddfrgchk` to analyse the output of the simulation. Finally, we have done throughput and actual seek measurements by observing the block I/O layer of the Linux kernel with Jens Axboe's `blktrace` [9].

All experiments were performed on a Linux 2.6.15.3 kernel, only modified with the `blktrace` patches. The PVR simulations on actual filesystems were running on fairly recent Pentium IV machines with a 250 GB Western Digital hard drive. The actual speed of the machine and hard drive should not influence the results of the fragmentation experiments, and the performance measurements were only compared with those conducted on the same machine.

4.1 LIMEFS

We isolated the simple allocation strategy of LIMEFS described in section 3.1.2 from the rest of the filesystem and implemented it in a simulation in user space. This way, the creation and deletion of files was performed by only modifying the in-memory meta data of the filesystem, and we were quickly able to investigate if our theory has practical value. The results of this experiment can be found in section 5.2.

4.2 `pvrsim`

As we clearly do not want to use a PVR for two years to see what the effects are of doing so, we have written a simulation program called `pvrsim`. This multi-threaded application writes and deletes files, similar in size to recorded broadcasts, as fast as possible. It is able to do so with a number of concurrent threads defined at runtime, to simulate the simultaneous recording of multiple streams.

The size of the generated file is uniformly distributed in the range from 500 MB to 5 GB. Besides that, every file is assigned a Gaussian distributed popularity score, ranging roughly from 10 to 100. This score is used to determine which file should be deleted to free up disk space for a new file.

When writing new files, `pvrsim` always keeps a minimum amount of free disk space to prevent excessive fragmentation. This dependency between fragmentation and free disk space was shown in (13) in section 3.3.3. If writing a new file would exceed this limit, older files are deleted until enough space has been freed. The file with the lowest weighed popularity is deleted first. The weighed popularity is determined by dividing the popularity by the logarithm of the age, where the age is expressed as the number of files created after the file at hand.

Blocks of 32 KB filled with zeroes are written to each file until it reaches the previously determined size. Next, the location of each block of the file is looked up and the extents of the file are determined. The extents are written to a log file for further processing by `hddfrgchk`.

4.3 `hddfrgchk`

Our main simulation tool `pvrsim` outputs the extents of each created file, which need further processing to be able to analyse the results properly. This processing is done by `hddfrgchk`, which provides two separate functions, described below.

4.3.1 Fragmentation measures

`hddfrgchk` is able to calculate a number of variables from the extent lists. First, it determines the number of fragments of which the file at hand consists. As explained in section 3.2, this number is often higher than the actual seeks the hard drive has to do when reading the file. We therefore calculate the theoretical number of seeks, based on the characteristics of a typical hard drive, with (3) from section 3.2. The exact parameters in this calculation were determined from a typical hard drive by measurements, as described in [10].

Furthermore, we have defined a measure for the relative effective data transfer speed. The minimum number of rotations the drive has to make to transfer all data of a file if all its blocks were contiguous can be calculated by dividing the number of blocks of the file by the average number of blocks on a track. The actual number of rotations the drive theoretically has to make to transfer the data can also be calculated. This is done by adding the blocks in the gaps that were not counted as fragments in our earlier calculations to the total amount of blocks in the file. Furthermore, the number of rotations that took place in the time the hard drive was seeking (the number of theoretical seeks as calculated earlier is used for this) is also added to the estimation of the number of rotations the drive has to make to transfer the file. Dividing the minimum number of rotations by the estimation of the actual number of rotations gives us the relative transfer speed.

4.3.2 Filesystem lay-out

Besides these variables, `hddfrgchk` also generates a graphical representation of the simulation over time. The filesystem is depicted by an image, each pixel representing a number of blocks. With each file written by the simulator, the blocks belonging to that file are given a separate colour. When the file is deleted, this is also updated in the image of the filesystem.

A new picture of the state of the filesystem is generated for every file, and the separate pictures are combined into an animation. The animation gives a visualisation of the locations of the files on the drive, and gives an insight on how the filesystem evolves.

4.4 Performance measurements

To verify the theoretical calculations of `hddfrgchk`, we also have done some measurements. We have looked at the requests issued to the block I/O device (the hard drive in this case) by the block I/O layer. The `blktrace` [9] patch by Jens Axboe provides useful instrumentation for this purpose in the kernel.

4.4.1 `blktrace`

The kernel-side mechanism collects request queue operations. The user space utility `blktrace` extracts those event traces via the Relay filesystem (RelayFS) [11]. The event traces are stored in a raw data format, to ensure fast processing. The `blkparse` utility produces formatted output of these traces afterwards, and generates statistics.

The events that are collected originate either from the file system or are SCSI commands. The filesystem block layer requests consist of the read or write actions of the filesystem. These actions are queued and inserted in the internal I/O scheduler queue. The requests might be merged with other items in the queue, at the discretion of the I/O scheduler. Subsequently, they are issued to the block device driver, which finally signals when a specific request is completed.

4.4.2 Deriving the number of seeks

All requests also include the Logical Block Address (LBA) of the starting sector of the request, as well as the size (in sectors) of the request. As we are interested in the exact actions the hard drive performs, we only look at the requests that are reported to be completed by the block device driver, along with their location and size. With this information, we can count the number of seeks the hard drive has made: if the starting location of a request is equal to the ending location of the previous request, no seek will take place. This does not yet account for the fact that small gaps might not induce seeks, but do lower the transfer rate.

4.4.3 Determining the data transfer rate

The effective data transfer rate can be derived by the information provided by `blktrace`, but can also be calculated just by the wall clock time needed to read a file, divided by the file size. Comparing transfer rates of various files should be done with caution: the physical location on the drive significantly influences this. To have a fair comparison, the average transfer rate over the whole drive should be compared at various stages in the simulation.

5 The results

We have conducted a number of variations of the simulations described in section 4. The variables under consideration were the filesystem on which the simulations were taking place, the size of the filesystem, the minimum amount of free space on the filesystem, the length of the simulation (i.e., the number of files created) and the number of concurrent files being written.

5.1 Simulation parameters

With exploratory simulations we discovered that the size of the filesystem is not of significant influence on the outcome of the experiments, as long as the filesystem is sufficiently large compared to both the block size and the average file size. As typical PVR systems typically offer storage space ranging from 100 GB to 300 GB, we decided on a filesystem size of 138 GB. Due to circumstances, however, some of the experiments were conducted on a filesystem of 100 GB.

The minimum amount of space that is always kept free is in the simulations with `pvrsim` fixed at 5% of the capacity of the drive. According to the preliminary in-memory LIMEFS experiments this is a reasonable value. The results of these experiments are elaborated in section 5.2.

The size of the created files is chosen randomly between 500 MB and 5 GB, uniformly distributed. As explained in section 2.1, these are typical file sizes for a PVR recording MPEG2 streams in Standard Definition (SD) resolution.

The length of the experiments, expressed in number of files created, was initially set at 10,000. The results from these runs showed that after about 2,500 files the behaviour stabilised. Therefore, the length of the simulations was set at 2,500 files.

We have done simulations with up to four simultaneous threads, so several files were written to the disk concurrently. This was done to observe the behaviour when recording multiple simultaneous broadcasts.

The filesystems we have covered in the experiments described in this paper are FAT (both Linux and Microsoft Windows), ext3 [2] [8], ReiserFS [12], LIMEFS and NTFS [13] (Microsoft Windows). We plan to cover more filesystems.

5.2 LIMEFS in-memory simulation

The results of the simulation of LIMEFS as described in section 4.1 are shown in figure 3. We have run our in-memory simulation on an imaginary 250 GB hard drive, with the minimum amount of free space as a variable parameter.

As can be seen, in the runs with 5%, 10% and 20% free space the fragmentation stabilises quickly. In longer runs we observed that the 0%, 1% and 2% options also stabilise, but the final fragmentation count is much higher and the stabilisation takes longer. The sweet spot appears to be 5% minimum free space, as this gives a good balance between fragmentation and hard drive space usage.

A nice result from this experiment is that the observed fragmentation counts fit our formula.

We have taken a filesystem of 250 GB and an average file size of 2750 MB. For the 5% free space run, this makes the fraction of free space $m = 0.05$ (see section 3.3.3). The fraction of the disk occupied by a file is $s = \frac{L}{N} = \frac{2.75}{250} = 0.01$. So, the number of fragments in a file on average, according to the formula, is:

$$L = \frac{1}{2m - s} = \frac{1}{2 \cdot 0.05 - 0.01} \approx 11$$

From the plot in figure 3, we can see the calculation matches the outcome of the simulation. The same holds for the other values for the minimum amount of free space.

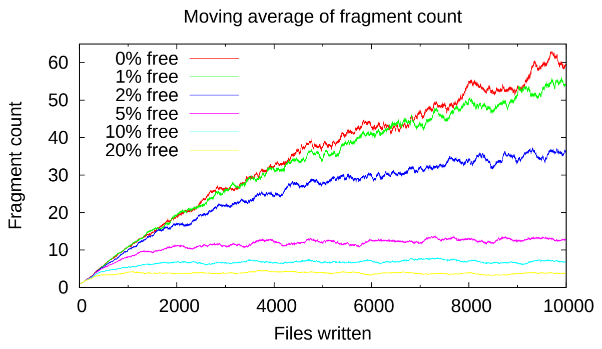


Figure 3: The average fragment count during a simulation run of 10,000 files, with a variable percentage of space that was kept free.

5.3 Fragmentation simulation

While the name `pvrsim` might suggest otherwise, we must stress that *all* results obtained by `pvrsim` were obtained by writing *real* files to *real* filesystems. The filesystems were running on their native platforms (*i.e.*, FAT and NTFS on Windows XP SP2, the others on Linux 2.6.15). For an interesting comparison however, we have also tested how the Linux version of FAT performs in a number of situations.

These experiments resulted in the plots in figure 4, where the number of seeks according to our theory (see section 3.2) and the relative speed are shown for single- and multi-threaded situations.

5.3.1 Single-threaded performance

All single-threaded simulations show similar results on all filesystems: the effective read speed is not severely impacted by writing many large files in succession. Some filesystems handle the fragmentation more gracefully than others, but the effects on system performance are negligible in all cases, as can be seen in the top two plots of figure 4. Although ext3 seems to have quite some fragmentation, the relative speed does not suffer: 98% of the raw data throughput is a hardly noticeable slowdown.

5.3.2 Multi-threaded performance

The multi-threaded simulations show that a file allocation strategy that tries to cluster files that are created concurrently performs considerably better compared to one that does not. The performance of NTFS deteriorates very quickly (after having written only a couple of files) to a relative speed of around 0.6, while the relative speed of ReiserFS and ext3 do not drop below 0.8. Linux FAT is doing slightly worse, while LIMEFS is not impacted at all.

5.3.3 LIMEFS

According to our results, the area of PVR applications is one where LIMEFS really shines. LIMEFS never produces more than around ten fragments, even with four threads. We do admit that LIMEFS is the only filesystem used in this experiment that was designed specifically for the purpose of storing PVR recordings, and that it might perform horribly or even not at all in other areas. However, the results are encouraging, and will hopefully serve as an inspiration for other filesystems.

5.3.4 FAT and NTFS

One interesting result of running the simulation on FAT and NTFS on Windows is that Windows appears to allocate 1 megabyte chunks regardless of the block size (extents are typically 16 clusters of 64k, or 32 clusters of 32k). As our simulation does not produce files smaller than 1 megabyte, we have no way of determining the effect of small files on the fragmentation levels of the filesystems. However, the chunked allocation seems to alleviate the negative effects of the otherwise quite naive allocation strategies of FAT and NTFS.

5.4 Seeks and throughput

We have measured the the number of the average data rate of files on a newly created filesystem and compared that with the data rate of files after `pvrsim` simulated a PVR workload, to confirm that our relative speed calculations are representative for the actual situation. Furthermore, we have analysed the activity in the block I/O layer with `blktrace` to see if the number of seeks derived from the placement of the blocks on the drive can be used to estimate the real activity.

The raw data throughput of the drive on which we have executed our single-threaded ext3 run was 60,590 KB/s. After writing 10,000 files with `pvrsim`, the data throughput while reading those files was on average 58,409 KB/s. This results in a relative speed of 0.96, which is close to the 0.95 we have estimated with our calculations. The figures of the two-threaded ext3 run on a different machine (52,520 KB/s raw data throughput, 40,270 KB/s while reading the final files present and thus a relative speed of 0.77, the same as calculated) confirm this.

With the use of `blktrace` we counted 10,267 seeks when reading the final files present on the disk after the single-threaded ext3 run. This is an average of 366 fragments per file. If we take into account that small fragments do not cause the drive to seek, as explained in section 3.2, the number of seeks caused by fragments after a gap of more than 676KB was, again according to the `blktrace` observations, 5692 or an average of 203 seeks per file. This is for all practical purposes close enough to the 198 seeks we derive from the location of the fragments on the disk.

6 Future work

The experiments and results presented in this paper are a starting point to improve the fragmentation robustness of filesystem for PVR-like scenarios. To obtain a good overview of the current state-of-the-art, we are planning to run our simulations on other filesystems, *e.g.*, on XFS [14]. We intend to cover more combinations of the parameters of our simulations as well, *e.g.*, different distributions for file size and popularity, different filesystem sizes, and different filesystem options.

Following this route, experimental measurements of different workloads and scenarios might provide interesting insights as well. We feel our tools could easily be modified to incorporate other synthetic workloads, and therefore be of great help for further experiments.

A more practical matter is improving the allocation strategy of the FAT filesystem. Making the allocation extent-based and multi-stream aware like LIMEFS will greatly improve the fragmentation behaviour, while the resulting filesystem will remain backwards compatible. On one hand, this proves our LIMEFS strategies in real-life applications, while on the other hand this will be useful for incorporation in mobile digital television devices, which might also act as a mass storage device and should therefore use a compatible filesystem. Unfortunately, the usefulness of FAT in a PVR context remains limited due to its file size limit of 4 gigabytes.

7 Conclusion

The formulae derived in 3 give an indication of the average fragmentation level for simple allocation strategies and large files. Although we made quite some assumptions and took some shortcuts in the mathematical justification, the results of the LIMEFS in-memory experiments of section 5.2 support the theory. Another useful outcome of the formulae is that, at least with large files, the fragmentation level stabilises, which seems to be true as well for filesystems with more sophisticated allocation strategies.

When dealing only with large files, a simple allocation strategy seems very efficient in terms of fragmentation prevention. Especially if only one stream is written, even FAT performs very well. Writing multiple streams simultaneously requires some precautions, but a strategy as implemented in LIMEFS suffices and outperforms all more complicated strategies with respect to the fragmentation level.

The ext3 and ReiserFS filesystems have a relatively high fragmentation in our scenario. However, the fragmentation stabilises and the impact is therefore predictable, and is no real issue with large files. A file of 2 GB consisting of 500 fragments will result in 4.5 seconds of seeking (with an average seek time of 9 ms). This is not significant for a movie of two hours, if the seeks are not clustered.

The relative speeds measured with ReiserFS and ext3 are not as good as the ones of LIMEFS, but still acceptable: 80% of the performance after prolonged use. NTFS however perform horribly when using multiple simultaneous streams. The Linux version of FAT is doing surprisingly well with two concurrent streams, much better than the Microsoft Windows implementation. We have still to investigate why this is.

An interesting observation is the fact that ext3 keeps about 2% of unused free space at the end of the drive, independent of the "reserved space" options (used to prevent the filesystem from being filled up by a normal user). If this free space is kept clustered at the end instead of being used throughout the simulation, this is inefficient in terms of fragmentation, as our formulae tell us.

In general, a PVR-like device is able to provide sustainable I/O performance over time if a filesystem like ext3 or ReiserFS is used. This does not assert anything about scenarios where file sizes are in the order of magnitude of the size of an allocation unit. However, the ratio between rotation time and seek time in modern hard drives is such that seeks are not something to avoid at all costs anymore. For optimal usage of the hard drive under a load of a number of concurrent streams, an allocation strategy that is aware of such a scenario is needed.

References

[1] Microsoft Corporation. *Microsoft Extensible Firmware Initiative FAT32*

File System Specification. Whitepaper, December 2000. `http://www.microsoft.com/whdc/system/platform/firmware/fatgendown.mspx?`

[2] Card, Rémy; Ts'o, Theodore; Tweedie, Stephen. *Design and Implementation of the Second Extended Filesystem*. Proceedings of the First Dutch International Symposium on Linux, 1994. `http://web.mit.edu/tytso/www/linux/ext2intro.html`

[3] MythTV. `http://www.mythtv.org`

[4] Microsoft Windows XP Media Center. `http://www.microsoft.com/windowsxp/mediacenter/default.mspx`

[5] Mesut, Özcan; Brink, Benno van den; Blijlevens, Jennifer; Bos, Eric; Nijs, Giel de. *Hard Disk Drive Power Management for Multi-stream Applications*. Proceedings of the International Workshop on Software Support for Portable Storage, March 2005.

[6] Nijs, Giel de; Almesberger, Werner; Brink, Benno van den. *Active Block I/O Scheduling System (ABISS)*. Proceedings of the Linux Symposium, vol. 1, pp. 109–126, Ottawa, July 2005. `http://www.linuxsymposium.org/2005/linuxsymposium_procv1.pdf`

[7] Springer, Rink. *Time is of the Essence: Implementation of the LimeFS Realtime Linux Filesystem*. Graduation Report, Fontys University of Applied Sciences, Eindhoven, 2005.

[8] Johnson, Michael K. *Red Hat's New Journaling File System: ext3*. Whitepaper, 2001. `http://www.redhat.com/support/wpapers/redhat/ext3/`

[9] Axboe, Jens; Brunelle, Alan D. *blktrace User Guide*. `http://www.kernel.org/pub/linux/kernel/people/axboe/blktrace/`

[10] Mesut, Özcan; Lambert, Niek. *HDD Characterization for A/V Streaming Applications*. IEEE Transactions on Consumer Electronics, Vol. 48, No. 3, 802–807, August 2002.

[11] Dagenais, Michel; Moore, Richard; Wisniewski, Bob; Yaghmour, Karim; Zanussi, Tom. *RelayFS - A High-Speed Data Relay Filesystem*. `http://relayfs.sourceforge.net/relayfs.txt`

[12] Reiser, Hans. *ReiserFS v.3 Whitepaper*. Whitepaper, 2003.

[13] Microsoft Corporation. *Local File Systems for Windows*. WinHEC, May 2004. `http://www.microsoft.com/whdc/device/storage/LocFileSys.mspx`

[14] Hellwig, Chrisoph. *XFS for Linux*. UKUUG, July 2003. `http://oss.sgi.com/projects/xfs/papers/ukuug2003.pdf`

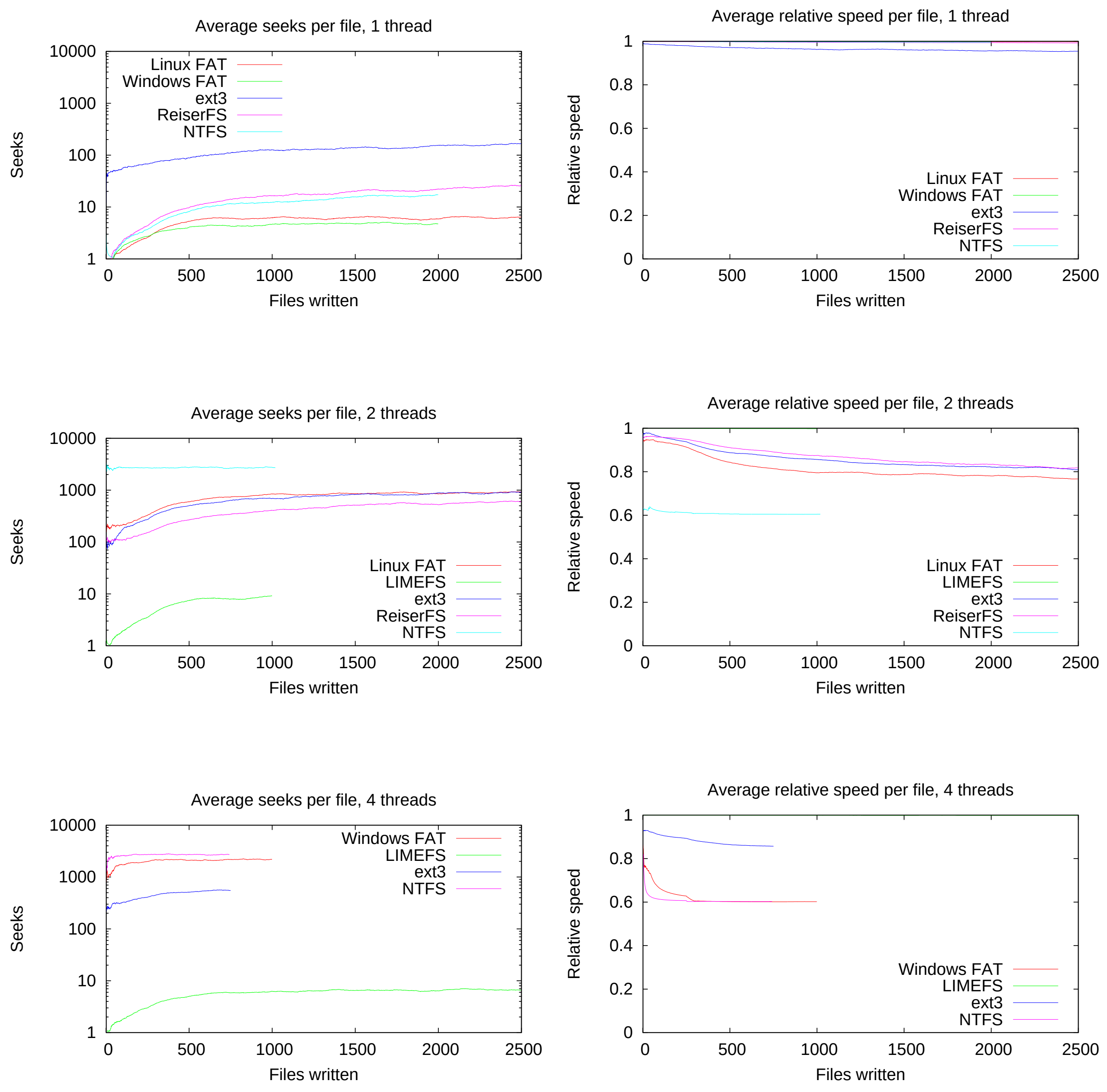


Figure 4: Results of `pvrsim` run on various filesystems. From top to bottom the number of concurrent threads was respectively one, two and four. The plots on the left side are the average amount of fragments over time, corrected to exclude small fragments as described in section 3.2. On the right side the relative speed (see section 4.3) is shown.

The LTTng tracer: A low impact performance and behavior monitor for GNU/Linux

Mathieu Desnoyers
École Polytechnique de Montréal
mathieu.desnoyers@polymtl.ca

Michel R. Dagenais
École Polytechnique de Montréal
michel.dagenais@polymtl.ca

Abstract

Efficient tracing of system-wide execution, allowing integrated analysis of both kernel space and user space, is something difficult to achieve. The following article will present you a new tracer core, Linux Trace Toolkit Next Generation (LTTng), that has taken over the previous version known as *LTT*. It has the same goals of low system disturbance and architecture independance while being fully reentrant, scalable, precise, extensible, modular and easy to use. For instance, LTTng allows tracepoints in NMI code, multiple simultaneous traces and a flight recorder mode. LTTng reuses and enhances the existing LTT instrumentation and RelayFS.

This paper will focus on the approaches taken by LTTng to fulfill these goals. It will present the modular architecture of the project. It will then explain how NMI reentrancy requires atomic operations for writing and RCU lists for tracing behavior control. It will show how these techniques are inherently scalable to multiprocessor systems. Then, time precision limitations in the kernel will be discussed, followed by an explanation of LTTng's own monotonic timestamps motives.

In addition, the template based code generator for architecture agnostic trace format will be presented. The approach taken to allow nested types, variable fields and dynamic alignment of data in the trace buffers will be revealed. It will show the mechanisms deployed to facilitate use and extension of this tool by adding custom instrumentation and analysis involving kernel, libraries and user space programs.

It will also introduce LTTng's trace analyzer and graphical viewer counterpart: Linux Trace Toolkit Viewer (LTTV). The latter implements extensible analysis of the trace information through collaborating text and graphical plugins.[1] It can simultaneously display multiple multi-GBytes traces of multi-processor systems.

1 Tracing goals

With the increasing complexity of newer computer systems, the overall performance of applications often depends on a combination of several factors including I/O subsystems, device drivers, interrupts, lock contention among multiple CPUs, scheduling and memory management. A low impact, high performance, tracing system may therefore be the only tool capable of collecting the information produced by instrumenting the whole system, while not

[1]Project website: http://ltt.polymtl.ca.

changing significantly the studied system behavior and performance.

Besides offering a flexible and easy to use interface to users, an efficient tracer must satisfy the requirements of the most demanding application. For instance, the widely used `printk` and `printf` statements are relatively easy to use and are correct for simple applications, but do not offer the needed performance for instrumenting interrupts in high performance multiprocessor computer systems and cannot necessarily be used in some code paths such as non maskable interrupts (NMI) handlers.

An important aspect of tracing, particularly in the real-time and high performance computing fields, is the precision of events timestamps. Real-time is often used in embedded systems which are based on a number of different architectures (e.g. ARM, MIPS, PPC) optimized for various applications. The challenge is therefore to obtain a tracer with precise timestamps, across multiple architectures, running from several MHz to several GHz, some being multi-processors.

The number of ad hoc tracing systems devised for specific needs (several Linux device drivers contain a small tracer), and the experience with earlier versions of LTT, show the needs for a flexible and extensible system. This is the case both in terms of adding easily new instrumentation points and in terms of adding plugins for the analysis and display of the resulting trace data.

2 Existing solutions

Several different approaches have been taken by performance monitoring tools. They usually adhere to one of the following two paradigms. The first class of monitor, *post-processing*, aims to minimize CPU usage during the execution of the monitored system by collecting data for later off-line analysis. As the goal is to have minimum impact on performance, static instrumentation is habitually used in this approach. Static instrumentation consists in modifying the program source code to add logging statements that will compile with the program. Such systems include LTT [7], a Linux kernel Tracer, K42 [5], a research operating system from IBM, IrixView and Tornado which are commercial proprietary products.

The second class of monitor aims at calculating well defined information (e.g. I/O requests per seconds, system calls per second per PID) on the monitored CPU itself: it is what is generally called a *pre-processing* approach. It is the case of SystemTAP [3], Kerninst [4], Sun's dtrace [1] and IBM's Performance and Environment Monitoring (PEM) [6]. All except PEM use a dynamic instrumentation approach. Dynamic instrumentation is performed by changing assembly instructions for breakpoints in the program binary objects loaded in memory, like the gdb debugger does. It is suitable to their goal because it generally has a negligible footprint compared to the pre-processing they do.

Since our goal is to support high performance and real-time embedded systems, the dynamic probe approach is too intrusive, as it implies using a costly breakpoint interrupt. Furthermore, even if the pre-processing of information can sometimes be faster than logging raw data, it does not allow the same flexibility as post-processing analysis. Indeed, almost every aspect of a system can be studied once is obtained a trace of the complete flow of the system behavior. However, pre-processed data can be logged into a tracer, as does PEM with K42, for later combined analysis, and the two are therefore not incompatible.

3 Previous Works

LTTng reuses research that has been previously done in the operating system tracing field in order to build new features and address currently unsolved questions more thoroughly.

The previous Linux Trace Toolkit (LTT) [7] project offers an operating system instrumentation that has been quite stable through the 2.6 Linux kernels. It also has the advantage of being cross-platform, but with types limited to fixed sizes (e.g. fixed 8, 16, 32, or 64-byte integers compared to host size byte, short, integer, and long). It also suffers from the monolithic implementation of both the LTT tracer and its viewer which have proven to be difficult to extend. Another limitation is the use of the kernel NTP corrected time for timestamps, which is not monotonic. LTTng is based on LTT but is a new generation, layered, easily extensible with new event types and viewer plugins, with a more precise time base and that will eventually support the combined analysis of several computers in a cluster [2].

RelayFS [8] has been developed as a standard high-speed data relay between the kernel and user space. It has been integrated in the 2.6.14 Linux kernels. It offers hooks for kernel clients to send information in large buffers and interacts with a user space daemon through file operations on a memory mapped file.

IBM, in the past years, has developed K42 [5], an open source research kernel which aims at full scalability. It has been designed from the ground up with tracing being a necessity, not an option. It offers a very elegant lockless tracing mechanism based on the atomic compare-and-exchange operation.

The Performance and Environment Monitoring (PEM) [6] project shares a few similarities with LTTng and LTTV since some work have been done in collaboration with members of their team. The XML file format for describing events came from these discussions, aiming at standardizing event description and trace formats.

4 The LTTng approach

The following subsections describe the five main components of the LTTng architecture. The first one explains the control of the different entities in LTTng. It is followed by a description of the data flow in the different modules of the application. The automated static instrumentation will thereafter be introduced. Event type registration, the mecanism that links the extensible instrumentation to the dynamic collection of traces, will then be presented.

4.1 Control

There are three main parts in LTTng: a user space command-line application, *lttctl*; a user space daemon, *lttd*, that waits for trace data and writes it to disk; and a kernel part that controls kernel tracing. Figure 1 shows the control paths in LTTng. *lttctl* is the command line application used to control tracing. It starts a *lttd* and controls kernel tracing behavior through a library-module bridge which uses a netlink socket.

The core module of LTTng is *ltt-core*. This module is responsible for a number of LTT control events. It controls helper modules *ltt-heartbeat*, *ltt-facilities*, and *ltt-statedump*. Module *ltt-heartbeat* generates periodic events in order to detect and account for cycle counters overflows, thus allowing a single monotonically increasing time base even if shorter 32-bit (instead of 64-bit) cycle counts are stored in each event. *Ltt-facilities* lists the facilities

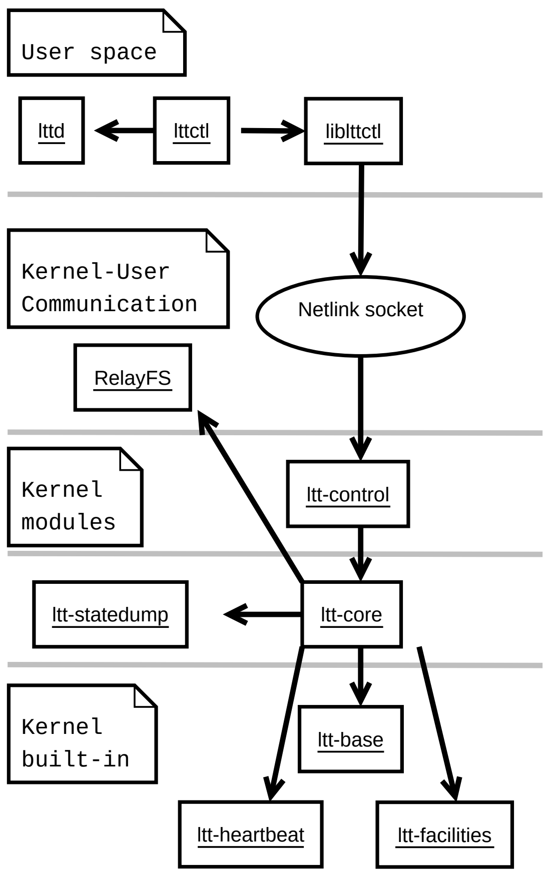


Figure 1: LTTng control architecture

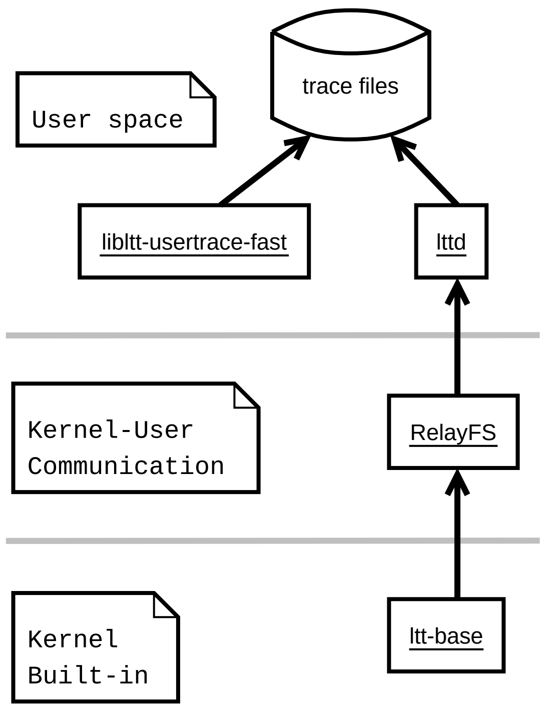


Figure 2: LTTng data flow

(collection of event types) currently loaded at trace start time. Module *ltt-statedump* generates events to describe the kernel state at trace start time (processes, files…). A builtin kernel object, *ltt-base*, contains the symbols and data structures required by builtin instrumentation. This includes principally the tracing control structures.

4.2 Data flow

Figure 2 shows the data flow in LTTng. All data is written through *ltt-base* into RelayFS circular buffers. When subbuffers are full, they are delivered to the *lttd* disk writer daemon.

Lttd is a standalone multithreaded daemon which waits on RelayFS channels (files) for data by using the poll file operation. When it is awakened, it locks the channels for reading by using a *relay buffer get* ioctl. At that point, it has exclusive access to the subbuffer it has reserved and can safely write it to disk. It should then issue a *relay buffer put* ioctl to release it so it can be reused.

A side-path, *libltt-usertrace-fast*, running completely in user space, has been developed for high throughput user space applications which need high performance tracing. It is explained in details in Section 4.5.4.

Both *lttd* and the *libltt-usertrace-fast* companion process currently support disk output, but should eventually be extended to other media like network communication.

4.3 Instrumentation

LTTng instrumentation, as presented in Figure 3, consists in an XML event description that

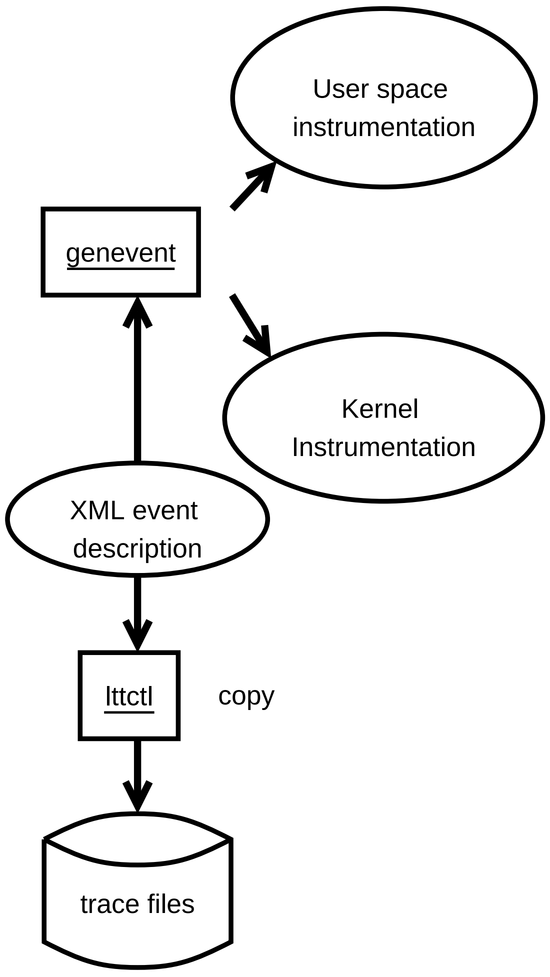


Figure 3: LTTng instrumentation

is used both for automatically generating *tracing headers* and as *data metainformation* in the trace files. These tracing headers implement the functions that must be called at instrumentation sites to log information in traces.

Most common types are supported in the XML description: fixed size integers, host size integers (int, long, pointer, size_t), floating point numbers, enumerations, and strings. All of these can be either host or network byte ordered. It also supports nested arrays, sequences, structures, and unions.

The tracing functions, generated in the tracing headers, serialize the C types given as arguments into the LTT trace format. This format supports both packed or aligned data types.

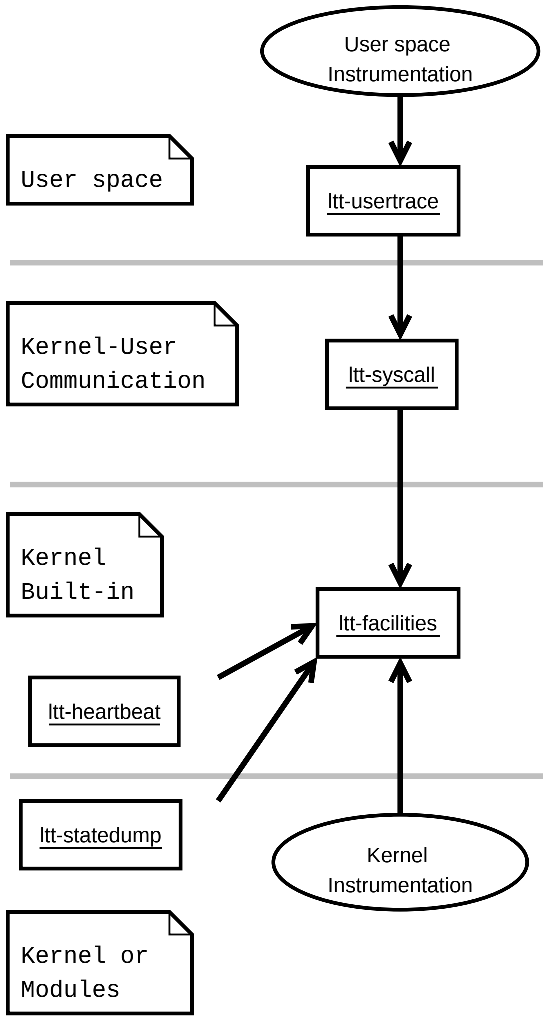


Figure 4: LTTng event type registration

A record generated by a probe hit is called an *event*. Event types are grouped in facilities. A *facility* is a dynamically loadable object, either a kernel module for kernel instrumentation or a user space library for user space instrumentation. An object that calls instrumentation should be linked with its associated facility object.

4.4 Event type registration

Event type registration is centralized in the *ltt-facilities* kernel object, as shown in Figure 4. It controls the rights to register specific type of information in traces. For instance, it does not allow a user space process using the *ltt-usertrace* API to register facilities with names conflicting with kernel facilities.

The *ltt-heartbeat* built-in object and the *ltt-statedump* also have their own instrumentation to log events. Therefore, they also register to *ltt-facilities*, just like standard kernel instrumentation.

Registered facility names, checksums and type sizes are locally stored in *ltt-facilities* so they can be dumped in a special low traffic channel at trace start. Dynamic registration of new facilities, while tracing is active, is also supported.

Facilities contain information concerning the type sizes in the compilation environment of the associated instrumentation. For instance, a facility for a 32-bit process would differ from the same facility compiled with a 64-bit process from its long and pointer sizes.

4.5 Tracing

There are many similarities between Figure 4 and Figure 5. Indeed, each traced information must have its metainformation registered into *ltt-facilities*. The difference is that Figure 4 shows how the metainformation is registered while Figure 5 show the actual tracing. The tracing path has the biggest impact on system behavior because it is called for every event.

Each event recorded uses *ltt-base*, container of the active traces, to get the pointers to RelayFS buffers. One exception is the *libltt-usertrace-fast* which will be explained at Subsection 4.5.4.

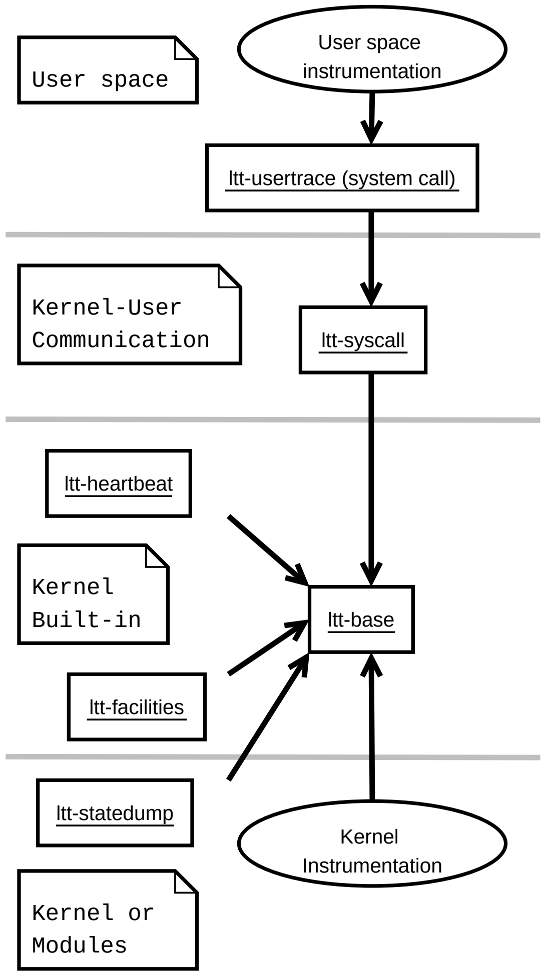


Figure 5: LTTng tracing

The algorithms used in these tracing sites which make them reentrant, scalable, and precise will now be explained.

4.5.1 Reentrancy

This section presents the lockless reentrancy mechanism used at LTTng instrumentation sites. Its primary goal is to provide correct tracing throughout the kernel, including non-

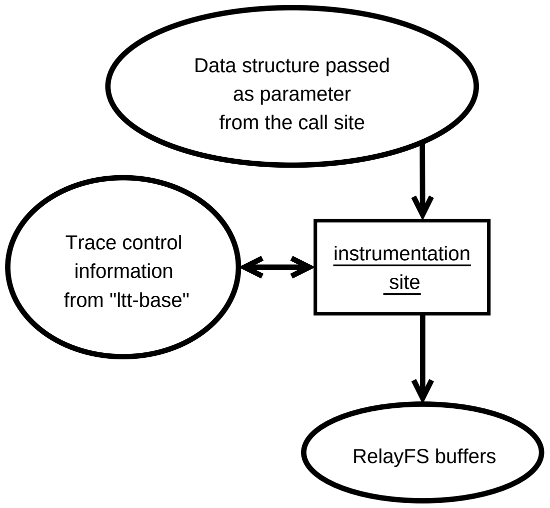

Data structure passed as parameter from the call site
Trace control information from "ltt-base"
instrumentation site
RelayFS buffers

Figure 6: LTTng instrumentation site

maskable interrupts (NMI) handlers which cannot be disabled like normal interrupts. The second goal is to have the minimum impact on performance by both having a fast code and not disrupting normal system behavior by taking intrusive locks or disabling interrupts.

To describe the reentrancy mechanism used by the LTTng instrumentation site (see Figure 6), we define the *call site*, which is the original code from the instrumented program where the tracing function is called. We also define the *instrumentation site*, which is the tracing function itself.

The instrumentation site found in the kernel and user space instrumentation has very well defined inputs and outputs. Its main input is the call site parameters. The call site must insure that the data given as parameter to the instrumentation site is properly protected with its associated locks. Very often, such data is already locked by the call site, so there is often no need to add supplementary locking.

The other input that the instrumentation site takes is the global trace control information. It is contained in a RCU list of active traces in the *ltt-base* object. Note that the instrumentation site uses the *trace control information* both as an input and output: this is both how tracing behavior is controlled and where variables that control writing to RelayFS buffers are stored.

The main output of the instrumentation site is a serialized memory write of both an event header and the instrumentation site parameters to the per-CPU RelayFS buffers. The location in these buffers is protected from concurrent access by using a lockless memory write scheme inspired from the one found in K42 [5]:

First, the *amount of memory space* necessary for the memory write is computed. When the data size is known statically, this step is quite fast. If, however, variable length data (string or sequence) must be recorded, a first size calculation pass is performed. *Alignment* of the data is taken care of in this step. To speed up data alignment, the start address of the variable size data is always aligned on the architecture size: it makes it possible to do a compile time alignement computation for all fixed size types.

Then, a memory region in the buffers is *reserved* atomically with a compare-and-exchange loop. The algorithm retries the reservation if a concurrent reserve occurs. The timestamp for the event is taken inside the compare-and-exchange loop so that it is monotically incrementing with buffer offsets. This is done to simplify data parsing in the post-processing tool.

A reservation can fail on the following conditions. In *normal* tracing mode, a buffer full condition causes the reservation to fail. On the other hand, in *flight recorder* mode, we overwrite non-read buffers, so it will never fail. When the reservation fails, the event lost counter is incremented and the instrumentation site will return without doing a *commit*.

The next step is to *copy* data from the instrumentation site arguments to the RelayFS reserved memory region. This step must preserve the same data alignment that has been calculated earlier.

Finally, a *commit* operation is done to release the reserved memory segment. No information is kept on a per memory region basis. We only keep a count of the number of reserved and committed bytes per subbuffer. A subbuffer is considered to be in a consistent state (non-corrupted and readable) when both counts are equal.

Is is possible that a process die between the slot reservation and commit because of a kernel OOPS. In that case, the lttd daemon will be incapable of reading the subbuffer affected by this condition because of unequal reserve and commit counts. This situation is resolved when the reservation algorithm wraps to the faulty subbuffer: if the reservation falls in a new buffer that has unequal reserve and commit counts, the reader (lttd) is pushed to the next subbuffer, the subbuffers lost counter is incremented, and the subbuffer is overwritten. To insure that this condition will not be reached by normal out of order commit of events (caused by nested execution contexts), the buffer must be big enough to contain data recorded by the maximum number of out of order events, which is limited by the longest sequence of events logged from nestable contexts (softirq, interrupts, and NMIs).

The *subbuffer delivery* is triggered by a flag from the call site on subbuffer switch. It is periodically checked by a timer routine to take the appopriate actions. This ensures atomicity and a correct lockless behavior when called from NMI handlers.

Compared to `printk`, which calls the scheduler, disables interrupts, and takes spinlocks, LTTng offers a more robust reentrancy that makes it callable from the scheduler code and from NMI handlers.

4.5.2 Scalability

Scalability of the tracing code for SMP machines is insured by use of per-CPU data and by the lockless tracing mechanism. The inputs of the instrumentation site are scalable: the data given as parameter is usually either on the caller's stack or already properly locked. The global trace information is organized in a RCU list which does not require any lock from the reader side.

Per-CPU buffers eliminate the false sharing of cachelines between multiple CPUs on the memory write side. The fact that input-output trace control structures are per-CPU also eliminates false sharing.

To identify more precisely the performance cost of this algorithm, let's compare two approaches: taking a per-CPU spinlock or using an atomic compare-and-exchange operation. The most frequent path implies either taking and releasing a spinlock along with disabling interrupts or doing a compare-and-exchange, an atomic increment of the reserve count, and an atomic increment of the commit count.

On a 3GHz Pentium 4, a compare-and-exchange without LOCK prefix costs 29 cycles. With a LOCK prefix, it raises to 112 cycles. An atomic increment costs respectively 7 and 93 cycles without and with a LOCK prefix. Using a spinlock with interrupts disabled costs 214 cycles.

As LTTng uses per-CPU buffers, it does not need to take a lock on memory to protect from other CPU concurrent access when performing these operations. Only the non-locked versions of compare-and-exchange and atomic increment are then necessary. If we consider only the time spent in atomic operations, using compare-and-exchange and atomic increments takes 43 cycles compared to 214 cycles for a spinlock.

Therefore, using atomic operations is five times faster than an equivalent spinlock on this architecture while having the additionnal benefit of being reentrant for NMI code and not disturbing the system behavior, as it does not disable interrupts for the duration of the tracing code.

4.5.3 Time (im)precision in the Linux kernel

Time precision in the Linux kernel is a research subject on its own. However, looking at the Linux kernel x86 timekeeping code is very enlightening on the nanosecond timestamps accuracy provided by the kernel. Effectively, it is based on a CPU cycle to nanosecond scaling factor computed at boot time based on the timer interrupt. The code that generates and uses this scaling factor takes for granted that the value should only be precise enough to keep track of scheduling periods. Therefore, the focus is to provide a fast computation of the time with shifting techniques more than providing a very accurate timestamp. Furthermore, doing integer arithmetic necessarily implies a loss of precision.

It causes problems when a tool like LTTng strongly depends on the monotonicity and precision of the time value associated with timestamps.

To overcome the inherent kernel time precision limitations, LTTng directly reads the CPU timestamp counters. It uses the `cpu_khz` kernel variable which contains the most precise calibrated CPU frequency available. This value will be used by the post-processing tool, LTTV, to convert cycles to nanoseconds in a precise manner with double precision numbers.

Due to the importance of the CPU timestamp counters in LTTng instrumentation, a workaround has been developed to support architectures that only have a 32-bit timestamp counter available. It uses the *ltt-heartbeat* module periodic timer to keep a full 64-bit timestamp counter on architectures where it is missing by detecting the 32-bit overflows in an atomic fashion; both the previous and the current TSC values are kept, swapped by a pointer change upon overflow. The read-side must additionnaly check for overflows.

It is important to restate that the time base used by LTTng is based neither on the kernel `do_gettimeofday`, which is NTP corrected and thus non monotonic nor on the kernel monotonic time, which suffers from integer arithmetic imprecision. LTTng uses the CPU timestamp counter and its most accurate calibration.

4.5.4 User space tracing

User space tracing has been achieved in many ways in the past. The original LTT [7] did use write operations in a device to send events to the kernel. It did not, however, give the same performances as in kernel events, as it needs a round-trip to the kernel and many copies of the information.

K42 [5] solves this by sharing per-CPU memory buffers between the kernel and user space processes. Although this is very performant, it does not insure secure tracing, as a given process can corrupt the traces that belong to other processes or to the kernel. Moreover, sharing memory regions between the kernel and user space might be acceptable for a research kernel, but for a production kernel, it implies a weaker traceability of process-kernel communications and might bring limitations on architectures with mixed 32- and 64-bit processes.

LTTng provides user space tracing through two different schemes to suit two distincts categories of instrumentation needs.

The first category is characterized by a very low event throughput. It can be the case of an event than happens rarely to show a specific error condition or periodically at an interval typically greater or equal to the scheduler period. The "slow tracing path" is targeted at this category.

The second category, which is addressed by the "fast tracing path," is much more demanding. It is particularly I/O intensive and must be close to the performance of a direct memory write. This is the case when instrumenting manually the critical path in a program or automatically every function entry/exit by gcc.

Both mecanisms share the same facility registration interface with the kernel, which passes through a system call, as shown in Figure 4. Validation is done by limiting these user space facilities to their own namespace so they cannot imitate kernel events.

The *slow path* uses a costly system call at each event call site. Its advantage is that it does not require linking the instrumented program against any library and does not have any thread startup performance impact like the *fast path* explained below. Every event logged through the system call is copied in the kernel tracing buffers. Before doing so, the system call verifies that the facility ID corresponds to a valid user space facility.

The *fast path*, *libltt-usertrace-fast* (at Figure 2) library consists in a per thread *companion* process which writes the buffers directly to disk. Communication between the thread and the library is done through the use of circular buffers in an anonymous shared memory map. Writing the buffers to disk is done by a separate *companion* process to insure that buffered data is never lost when the traced program terminates. The other goal is to account the time spent writing to disk to a different process than the one being traced. The file is written in the filesystem, arbitrarily in `/tmp/ltt-usertrace`, in files following this naming convention: `process-tid-pid-timestamp`, which makes it unique for the trace. When tracing is over, the `/tmp/ltt-usertrace` must be manually moved into the kernel trace. The trace and usertrace do not have to coincide: although it is better to have the usertrace time span included in the kernel trace interval to benefit from the scheduler information for the running processes, it is not mandatory and partial information will remain available.

Both the slow and the fast path reuse the lockless tracing algorithm found in the LTTng kernel tracer. In the fast path, it ensures reentrancy with signal handlers without the cost of disabling signals at each instrumentation site.

5 Graphical viewer: LTTV

LTTng is independent of the viewer, the trace format is well documented and a trace-reading library is provided. Nonetheless, the associated viewer, LTTV, will be briefly introduced. It implements optimised algorithms for random access of several multi-GB traces, describing the behavior of one or several uniprocessor or multi-processor systems. Many plugin views can be loaded dynamically into LTTV for the display and analysis of the data. Developers can thus easily extend the tool by creating their own instrumentation with the flexible XML description and connect their own plugin to that information. It is layered in a modular architecture.

On top of the LGPL low-level trace files reading library, LTTV recreates its own representation of the evolving kernel state through time and keeps statistical information into a generic hierarchical container. By combining the kernel state, the statistics, and the trace events, the viewers and analysis plugins can extend the information shown to the user. Plugins are kept focused (analysis, text, or graphical display, control...) to increase modularity and reuse. The plugin loader supports dependency control.

LTTV also offers a rich and performant event filter, which allows specifying, with a logical expression, the events a user is interested to see. It can be reused by the plugins to limit their scope to a subset of the information.

For performance reasons, LTTV is written in C. It uses the GTK graphical library and glib. It is distributed under the GPLv2 license.

6 Results

This section presents the results of several measurements. We first present the time overhead on the system running microbenchmarks of the instrumentation site. Then, taking these results as a starting point, the interrupt and scheduler impact will be discussed. Macrobenchmarks of the system under different loads will then be shown, detailing the time used for tracing.

The size of the instrumentation object code will be discussed along with possible size optimisations. Finally, time precision calibration is performed with a NMI timer.

6.1 Test environment

The test environment consists of a 3GHz, uniprocessor Pentium 4, with hyperthreading disabled, running LTTng 0.5.41. The results are presented in cycles; the exact calibration of the CPU clock is 3,000.607 MHz.

6.2 Microbenchmarks

Table 1 presents probe site microbenchmarks. Kernel probe tests are done in a kernel module with interrupts disabled. User space tests are influenced by interrupts and the scheduler. Both consist in 20,000 hits of a probe that writes 4 bytes plus the event header (20 bytes). Each hit is surrounded by two timestamp counter reads.

When compiling out the LTTng tracing, calibration of the tests shows that the time spent in the two TSC reads varies between 97 and 105 cycles, with an average of 100.0 cycles. We therefore removed this time from the raw probe time results.

As we can see, the best case for kernel tracing is a little slower than the *ltt-usertrace-fast* library: this is due to supplementary operations that must be done in the kernel (preemption disabling for instance) that are not needed in user space. The maximum and average values of time spent in user space probes does not mean much because they are sensitive to scheduling and interrupts.

The key result in Table 1 is the average 288.5 cycles (96.15ns) spent in a probe.

LTTng probe sites do not increase latency because they do not disable interrupts. However, the interrupt entry/exit instrumentation itself does increase interrupt response time, which

Probe site	Test Series	Time spent in probe (cycles)		
		min	average	max
Kernel	Tracing dynamically disabled	0	0.000	338
Kernel	Tracing active (1 trace)	278	288.500	6,997
User space	*ltt-usertrace-fast* library	225	297.021	88,913
User space	Tracing through system call	1,013	1,042.200	329,062

Table 1: LTTng microbenchmarks for a 4-byte event probe hit 20,000 times

therefore increases low priority interrupts latency by twice the probe time, which is 577.0 cycles (192.29ns).

The scheduler response time is also affected by LTTng instrumentation because it must disable preemption around the RCU list used for control. Furthermore, the scheduler instrumentation itself adds a task switch delay equal to the probe time, for a total scheduler delay of twice the probe time: 577.0 cycles (192.29ns). In addition, a small implementation detail (use of `preempt_enable_no_resched()`), to insure scheduler instrumentation reentrancy, has a downside: it can possibly make the scheduler miss a timer interrupt. This could be solved for real-time applications by using the *no resched* flavour of preemption enabling only in the scheduler, wakeup, and NMI nested probe sites.

6.3 Macrobenchmarks

6.3.1 Kernel tracing

Table 2 details the time spent both in the instrumentation site and in *lttd* for different loads. Time spent in instrumentation is computed from the average probe time (288.5 cycles) multiplied by the number of probe hits. Time spent in *lttd* is the CPU time of the *lttd* process as given in the LTTV analysis. The load is computed by subtracting the time spent in *system call* mode in process 0 (idle process) from the wall time.

It is quite understandable that the probes triggered by the ping flood takes that much CPU time, as it instruments a code path that is called very often: the *system call* entry. The total cpu time used by tracing on a busy system (medium and high load scenarios) goes from 1.54 to 2.28%.

6.3.2 User space tracing

Table 3 compares the *ltt-usertrace-fast* user space tracer with gprof on a specific task: the instrumentation of each function entry and exit of a gcc compilation execution. You will see that the userspace tracing of LTTng is only a constant factor of 2 slower than a gprof instrumented binary, which is not bad considering the amount of additional data generated. The factor of 2 is for the ideal case where the daemon writes to a `/dev/null` output. In practice, the I/O device can further limit the throughput. For instance, writing the trace to a SATA disk, LTTng is 4.13 slower than gprof.

The next test consists in running an instrumented version of gcc, itself compiled with option `-finstrument-functions`, to compile a 6.9KiB C file into a 15KiB object, with level 2 optimisation.

As Table 3 shows, gprof instrumented gcc takes 1.73 times the normal execution time. The

Load size	Test Series	CPU time (%)			Data rate (MiB/s)	Events/s
		load	probes	lttd		
Small	mozilla (browsing)	1.15	0.053	0.27	0.19	5,476
Medium	find	15.38	1.150	0.39	2.28	120,282
High	find + gcc	63.79	1.720	0.56	3.24	179,255
Very high	find + gcc + ping flood	98.60	8.500	0.96	16.17	884,545

Table 2: LTTng macrobenchmarks for different loads

gcc instrumentation	Time (s)	Data rate (MiB/s)
not instrumented	0.446	
gprof	0.774	
LTTng (null output)	1.553	153.25
LTTng (disk output)	3.197	74.44

Table 3: gcc function entry/exit tracing

Instrumentation	object code size (bytes)
log 4-byte integer	2,288
log variable length string	2,384
log a structure of int, string, sequence of 8-byte integers	2,432

Table 4: Instrumentation object size

fast userspace instrumentation of LTTng is 3.22 times slower than normal. Gprof only extracts sampling of function time by using a periodical timer and keeps per function counters. LTTng extracts the complete function call trace of a program, which generates an output of 238MiB in 1.553 seconds (153.25 MiB/s). The execution time is I/O-bound, it slows down to 3.197s when writing the trace on a SATA disk through the operating system buffers (74.44 MiB/s).

6.4 Instrumentation objects size

Another important aspect of instrumentation is the size of the binary instructions added to the programs. This wastes precious L1 cache space and grows the overall object code size, which is more problematic in embedded systems. Table 4 shows the size of stripped objects that only contain intrumentation.

Independently of the amount of data to trace, the object code size only varies in our tests of a maximum of 3.3% from the average size. Adding 2.37kB per event might be too much for embedded applications, but a tradeoff can be done between inlining of tracing sites (and reference locality) and doing function calls, which would permit instrumentation code reuse.

A complete L1 cache hit profiling should be done to fully see the cache impact of the instrumentation and help tweak the inlining level. Such profiling is planned.

6.5 Time precision

Time precision measurement of a timestamp counter based clock source can only be done relatively to another clock source. The following test traces the NMI watchdog timer, using it as a comparison clock source. It has the advantage of not being disturbed by CPU load as these interruptions cannot be deactivated. It is, however, limited by the precision of

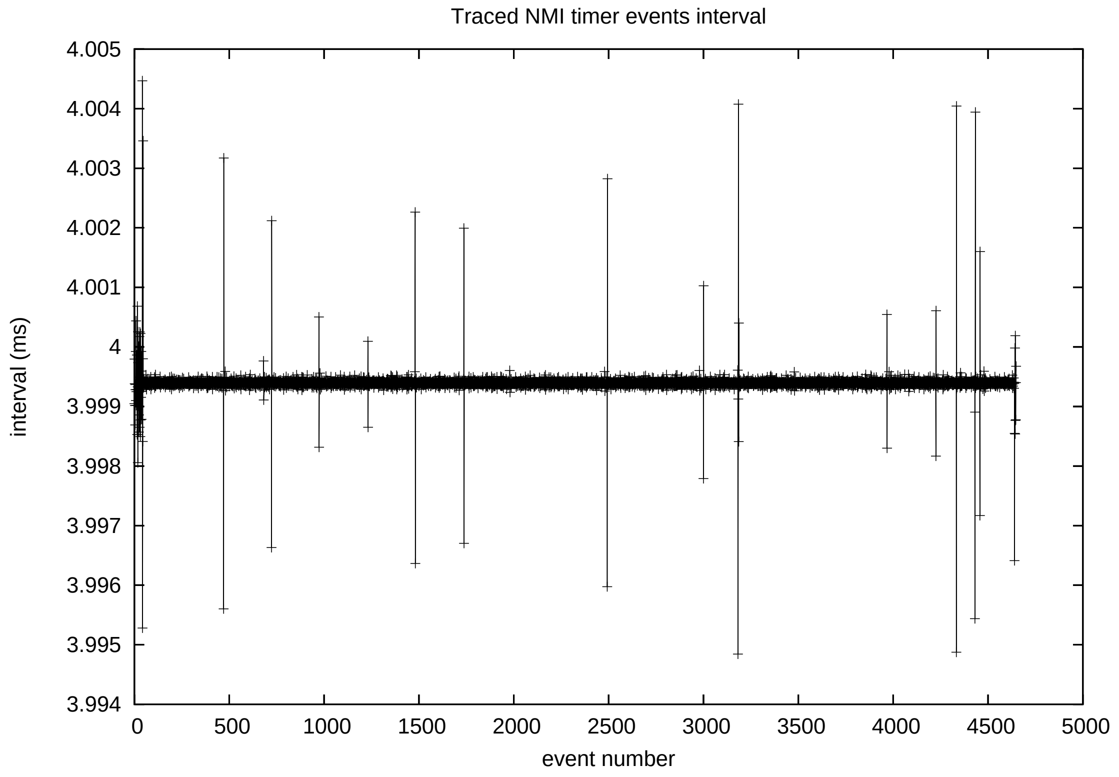


Figure 7: Traced NMI timer events interval

the timer crystal. The hardware used for these tests is an Intel D915-GAG motherboard. Its timer is driven by a TXC HC-49S crystal with a ±30 PPM precision. Table 5 and Figure 7 show the intervals of the logged NMI timer events. Their precision is discussed.

Statistic	value (ns)
min	3,994,844
average	3,999,339
max	4,004,468
standard deviation	52
max deviation	5,075

Table 5: Traced NMI timer events interval

This table indicates a standard deviation of 52ns and a maximum deviation of 5,075ns from the average. If we take the maximum deviation as a worse case, we can assume than we have a ±5.075μs error between the programmable interrupt timer (PIT) and the trace time base (derived from the CPU TSC). Part of it is due to the CPU cache misses, higher priority NMIs, kernel minor page faults, and the PIT itself. A 52ns standard deviation each 4ms means a 13μs error each second, for a 13 PPM frequency precision which is within the expected limits.

7 Conclusion

As demonstrated in the previous section, LTTng is a low disturbance tracer that uses about 2% of CPU time on a heavy workload. It is entirely based on atomic operations to insure reentrancy. This enables it to trace a wide range of code sites, from user space programs and libraries to kernel code, in every execution context, including NMI handlers.

Its time measurement precision gives a 13 PPM frequency error when reading the programmable interrupt timer (PIT) in NMI mode, which is coherent with the 30 PPM crystal precision.

LTTng proves to be a performant and precise tracer. It offers an architecture independent instrumentation code generator, from templates, to reduce instrumentation effort. It provides efficient and convenient mechanisms for kernel and user space tracing.

A plugin based analysis tool, LTTV, helps to further reduce the effort for analysis and visualisation of complex operating system behavior. Some work is actually being done in time synchronisation between cluster nodes, to extend LTTV to cluster wide analysis.

You are encouraged to use this tool and create new instrumentations, either in user space or in the kernel. LTTng and LTTV are distributed under the GPLv2 license.[2]

References

[1] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *USENIX '04*, 2004.

[2] Michel Dagenais, Richard Moore, Robert Wisniewski, Karim Yaghmour, and Thomas Zanussi. Efficient and accurate tracing of events in linux clusters. In *Proceedings of the Conference on High Performance Computing Systems (HPCS)*, 2003.

[3] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Brad Chen. Locating system problems using dynamic instrumentation. In *OLS (Ottawa Linux Symposium) 2005*, 2005.

[4] Ariel Tamches and Barton P. Miller. Fine-grained dynamic instrumentation of commodity operating system kernels. In *3rd Symposium on Operating Systems Design and Implementation*, February 1999.

[5] Robert W. Wisniewski and Bryan Rosenburg. Efficient, unified, and scalable performance monitoring for multiprocessor operating systems. In *Supercomputing, 2003 ACM/IEEE Conference*, 2003.

[6] Robert W. Wisniewski, Peter F. Sweeney, Kartik Sudeep, Matthias Hauswirth, Evelyn Duesterwald, Calin Cascaval, and Reza Azimi. Pem performance and environment monitoring for whole-system characterization and optimization. In *PAC2 (Conference on Power/Performance interaction with Architecture, Circuits, and Compilers)*, 2004.

[7] Karim Yaghmour and Michel R. Dagenais. The linux trace toolkit. *Linux Journal*, May 2000.

[8] Tom Zanussi, Karim Yaghmour Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for transmitting data from kernel to user space. In *OLS (Ottawa Linux Symposium) 2003*, pages 519–531, 2003.

[2]Project website: `http://ltt.polymtl.ca`

Linux as a Hypervisor

An Update

Jeff Dike
Intel Corp.
jeffrey.g.dike@intel.com

Abstract

Virtual machines are a relatively new workload for Linux. As with other new types of applications, Linux support was somewhat lacking at first and improved over time.

This paper describes the evolution of hypervisor support within the Linux kernel, the specific capabilities which make a difference to virtual machines, and how they have improved over time. Some of these capabilities, such as `ptrace` are very specific to virtualization. Others, such as AIO and `O_DIRECT` support help applications other than virtual machines.

We describe areas where improvements have been made and are mature, where work is ongoing, and finally, where there are currently unsolved problems.

1 Introduction

Through its history, the Linux kernel has had increasing demands placed on it as it supported new applications and new workloads. A relatively new demand is to act as a hypervisor, as virtualization has become increasingly popular. In the past, there were many weaknesses in the ability of Linux to be a hypervisor. Today, there are noticeably fewer, but they still exist.

Not all virtualization technologies stress the capabilities of the kernel in new ways. There are those, such as qemu, which are instruction emulators. These don't stress the kernel capabilities—rather they are CPU-intensive and benefit from faster CPUs rather than more capable kernels. Others employ a customized hypervisor, which is often a modified Linux kernel. This will likely be a fine hypervisor, but that doesn't benefit the Linux kernel because the modifications aren't pushed into mainline.

User-mode Linux (UML) is the only prominent example of a virtualization technology which uses the capabilities of a stock Linux kernel. As such, UML has been the main impetus for improving the ability of Linux to be a hypervisor. A number of new capabilities have resulted in part from this, some of which have been merged and some of which haven't. Many of these capabilities have utility beyond virtualization, as they have also been pushed by people who are interested in applications that are unrelated to virtualization.

`ptrace` is the mechanism for virtualizing system calls, and is the core of UML's virtualization of the kernel. As such, some changes to `ptrace` have improved (and in one case, enabled) the ability to virtualize Linux.

Changes to the I/O system have also improved the ability of Linux to support guests. These were driven by applications other than virtualization, demonstrating that what's good for virtualization is often good for other workloads as well.

From a virtualization point of view, AIO and `O_DIRECT` allow a guest to do I/O as the host kernel does—straight to the disk, with no caching between its own cache and the device. In contrast, `MADV_REMOVE` allows a guest to do something which is very difficult for a physical machine, which is to implement hotplug memory, by releasing pages from the middle of a mapped file that's backing the guest's physical memory.

FUSE (Filesystems in Userspace), another recent addition, is also interesting, this time from a manageability standpoint. This allows a guest to export its filesystem to the host, where a host administrator can perform some guest management tasks without needing to log in to the guest.

There is a new effort to add a virtualization infrastructure to the kernel. A number of projects are contributing to this effort, including OpenVZ, vserver, UML, and others who are more interested in resource control than virtualization. This holds the promise of allowing guests to achieve near-native performance by allowing guest process system calls to execute on the host rather than be intercepted and virtualized by `ptrace`.

Finally, there are a few problem areas which are important to virtualization for which there are no immediate solutions. It would be convenient to be able to create and manage address spaces separately from processes. This is part of the UML SKAS host patch, but the mechanism implemented there won't be merged into mainline. The current virtualization infrastructure effort notwithstanding, system call interception will be needed for some time to come. So, system call interception will still be an area of concern. Ingo Molnar implemented a mechanism called VCPU which effectively allows a process to intercept its own system calls. This hasn't been looked at in any detail, so it's too early to see if this is a better way for virtual machines to do system call interception.

2 The past

2.1 `ptrace`

When UML was first introduced, Linux was incapable of acting as a hypervisor[1]. `ptrace` allows one process to intercept the system calls of another both at system call entry and exit. The tracing process can examine and modify the registers of the traced child. For example, `strace` simply examines the process registers in order to print the system call, its arguments, and return value. Other tools, UML included, modify the registers in order to change the system call arguments or return value. Initially, on i386, it was impossible to change the actual system call, as the system call number had already been saved before the tracing parent was notified of the system call. UML needed this in order to nullify system calls so that they would execute in such way as to cause no effects on the host. This was done by changing the system call to `getpid`. A patch to fix this was developed soon after UML's first release, and it was fairly quickly accepted by Linus.

While this was a problem on i386, architectures differ on their handling of attempts to change system call numbers. The other architectures to which UML has been ported (x86_64, s390, and ppc) all handled this correctly, and needed

[1]on i386, which was the only platform UML ran on at the time

no changes to their system call interception in order to run UML.

Once ptrace was capable of supporting UML, attention turned to its performance, as virtualized system calls are many times slower than non-virtualized ones. An intercepted system call involves the system call itself, plus four context switches—to the parent and back on both system call entry and exit. UML, and any other tool which nullifies and emulates system calls, has no need to intercept the system call exit. So, another ptrace patch, from Laurent Vivier, added PTRACE_SYSEMU, which causes only system call entry to notify the parent. There is no notification on system call exit. This reduces the context switching due to system call interception by 50%, with a corresponding performance improvement for benchmarks that execute a system call in a tight loop. There is also a noticeable performance increase for workloads that are not system call-intensive. For example, I have measured a ~3% improvement on a kernel build.

2.2 AIO and O_DIRECT

While these ptrace enhancements were driven solely by the needs of UML, most of the other enhancements to the kernel which make it more capable as a hypervisor were driven by other applications. This is the case of the I/O enhancements, AIO and O_DIRECT, which had been desired by database vendors for quite a while.

AIO (Asynchronous IO) is the ability to issue an I/O request without having to wait for it to finish. The familiar read and write interfaces are synchronous—the caller can use them to make one I/O request and has to wait until it finishes before it can make another request. The wait can be long if the I/O requires disk access, which hurts the performance of processes which could have issued more requests or done other work in the meantime

A virtual OS is one such process. The kernel typically issues many disk I/O requests at a time, for example, in order to perform readahead or to swap out unused memory. When these requests are performed sequentially, as with read and write, there is a large performance loss compared to issuing them simultaneously. For a long time, UML handled this problem by using a separate dedicated thread for I/O. This allowed UML to do other work while an I/O request was pending, but it didn't allow multiple outstanding I/O requests.

The AIO capabilities which were introduced in the 2.6 kernel series do allow this. On a 2.6 host, UML will issue many requests at once, making it act more like a native kernel.

A related capability is O_DIRECT I/O. This allows uncached I/O—the data isn't cached in the kernel's page cache. Unlike a cached write, which is considered finished when the data is stored in the page cache, an O_DIRECT write isn't completed until the data is on disk. Similarly, an O_DIRECT read brings the data in from disk, even if it is available in the page cache. The value of this is that it allows processes to control their own caching without the kernel performing duplicate caching on its own. For a virtual machine, which comes with its own caching system, this allows it to behave like a native kernel and avoid the memory consumption caused by buffered I/O.

2.3 MADV_REMOVE

Unlike AIO and O_DIRECT, which allow a virtual kernel to act like a native kernel, MADV_REMOVE allows it implement hotplug memory, which is very much more difficult for a physical machine. UML implements its physical memory by creating a file on the host of the appropriate size and mapping pages from it into its own address space and those of its processes. I have long wanted a way to be able to free dirty pages from this file to the host as though they were clean. This would allow a simple way to manage the host's memory by moving it between virtual machines.

Removing memory from a virtual machine is done by allocating pages within it and freeing those pages to the host. Conversely, adding memory is done by freeing previously allocated pages back to the virtual machine's VM system. However, if dirty pages can't be freed on the host, there is no benefit.

I implemented one mechanism for doing this some time ago. It was a new driver, `/dev/anon`, which was based on tmpfs. UML physical memory is formed by mapping this device, which has the semantics that when a page is no longer mapped, it is freed. With `/dev/anon`, in order to pull memory from a UML instance, it is allocated from the guest VM system and the corresponding `/dev/anon` pages are unmapped. Those pages are freed on the host, and another instance can have a similar amount of memory plugged in.

This driver was never seriously considered for submission to mainline because it was a fairly dirty kludge to the tmpfs driver and because it was never fully debugged. However, the need for something equivalent remained.

Late in 2005, Badari Pulavarty from IBM proposed an `madvise` extension to do something equivalent. His motivation was that some IBM database wanted better control over its memory consumption and needed to be able to poke holes in a tmpfs file that it mapped. This is exactly what UML needed, and Hugh Dickens, who was aware of my desire for this, pointed Badari in my direction. I implemented a memory hotplug driver for UML, and he used it in order to test and debug his implementation.

`MADV_REMOVE` is now in mainline, and at this writing, the UML memory hotplug driver is in -mm and will be included in 2.6.17.

3 Present

3.1 FUSE

FUSE[2] is an interesting new addition to the kernel. It allows a filesystem to be implemented by a userspace driver and mounted like any in-kernel filesystem. It implements a device, `/dev/fuse`, which the userspace driver opens and uses to communicate with the kernel side of FUSE. It also implements a filesystem, with methods that communicate with the driver. FUSE has been used to implement things like sshfs, which allows filesystem access to a remote system over ssh, and ftpfs, which allows an ftp server to be mounted and accessed as a filesystem.

UML uses FUSE to export its filesystem to the host. It does so by translating FUSE requests from the host into calls into its own VFS. There were some mismatches between the interface provided by FUSE and the interface expected by the UML kernel. The most serious was the inability of the `/dev/fuse` device to support asynchronous operation—it didn't support `O_ASYNC` or `O_NONBLOCK`. The UML kernel, like any OS kernel, is event-driven, and works most naturally when requests and other things that require attention generate interrupts. It must also be possible to tell when a particular interrupt source is empty. For a file, this means that when it is read, it returns `-EAGAIN`

[2]http://fuse.sourceforge.net/

instead of blocking when there is no input available. `/dev/fuse` didn't do either, so I implemented both `O_ASYNC` and `O_NONBLOCK` support and sent the patches to Miklos Szeredi, the FUSE maintainer.

The benefit of exporting a UML filesystem to the host using FUSE is that it allows a number of UML management tasks to be performed on the host without needing to log in to the UML instance. For example, it would allow the host administrator to reset a forgotten root password. In this case, root access to the UML instance would be difficult, and would likely require shutting the instance down to single-user mode.

By chrooting to the UML filesystem mount on the host, the host admin can also examine the state of the instance. Because of the chroot, system tools such as ps and top will see the UML `/proc` and `/sys`, and will display the state of the UML instance. Obviously, this only provides read access to this state. Attempting to kill a runaway UML process from within this chroot will only affect whatever host process has that process ID.

3.2 Kernel virtualization infrastructure

There has been a recent movement to introduce a fairly generic virtualization infrastructure into the kernel. Several things seemed to have happened at about the same time in order to make this happen. Two virtualization projects, Virtuozzo and vserver, which had long maintained their kernel changes outside the mainline kernel tree, expressed an interest in getting their work merged into mainline. There was also interest in related areas, such as workload migration and resource management.

This effort is headed in the direction of introducing namespaces for all global kernel data. The concept is the same as the current filesystem namespaces—processes are in the global namespace by default, but they can place themselves in a new namespace, at which point changes that they make to the filesystem aren't visible to processes outside the new namespace. The changes in question are changed mounts, not changed files—when a process in a new namespace changes a file, that's visible outside the namespace, but when it make a mount in its namespace, that's not visible outside. For filesystems, the situation is more complicated than that because there are rules for propagating new mounts between namespaces. However, for virtualization purposes, the simplest view of namespaces works—that changes within the namespace aren't visible outside it.

When[3] finished, it will be possible to create new instantiations of all of the kernel subsystems. At this point, virtualization approaches like OpenVZ and vserver will map pretty directly onto this infrastructure.

UML will be able to put this to good use, but in a different way. It will allow UML to have its process system calls run directly on the host, without needing to intercept and emulate them itself. UML will create an virtualized instance of a subsystem, and configure it as appropriate. At that point, UML process system calls which use that subsystem can run directly on the host and will behave the same as if it had been executed within UML.

For example, virtualizing time will be a matter of introducing a time namespace which contains an offset from the host time. Any process within this namespace will see a system time that's different from the host time by the amount of this offset. The offset is changed by `settimeofday`, which can now be an unprivileged operation since its effects are invisible outside the time namespace.

[3]or if—some subsystems will be difficult to virtualize

gettimeofday will take the host time and add the namespace offset, if any.

With the time namespace working, UML can take advantage of it by allowing gettimeofday to run directly on the host without being intercepted. settimeofday will still need to be intercepted because it will be a privileged operation within the UML instance. In order to allow it to run on the host, user and groups IDs will need to be virtualized as well.

UML will be able to use the virtualized subsystems as they become available, and not have to wait until the infrastructure is finished. To do this, another ptrace extension will be needed. It will be necessary to selectively intercept system calls, so a system call mask will be added. This mask will specify which system calls should continue to be intercepted and which should be allowed to execute on the host.

Since some system calls will sleep when they are executed on the host, the UML kernel will need to be notified. When a process sleeps in a system call, UML will need to schedule another process to run, just as it does when a system call sleeps inside UML. Conversely, when the host system call continues running, the UML will need to be notified so that it can mark the process as runnable within its own scheduler. So, another ptrace extension, asking for notification when a child voluntarily sleeps and when it wakes up again, will be needed. As a side-benefit, this will also provide notification to the UML kernel when a process sleeps because it needs a page of memory to be read in, either because that page hadn't been loaded yet or because it had been swapped out. This will allow UML to schedule another process, letting it do some work while the first process has its page fault handled.

3.3 remap_file_pages

When page faults are virtualized, they are fixed by calling either mmap or mprotect[4] on the host. In the case of mapping a new page, a new vm_area_struct (VMA) will be created on the host. Normally, a VMA describes a large number of contiguous pages, such as the process text or data regions, being mapped from a file into a region of a process virtual memory.

However, when page faults are virtualized, as with UML, each host VMA covers a single page, and a large UML process can have thousands of VMAs. This is a performance problem, which Ingo Molnar solved by allowing pages to be rearranged within a VMA. This is done by introducing a new system call, remap_file_pages, which enables pages to be mapped without creating a new VMA for each one. Instead, a single large mapping of the file is created, resulting in a single VMA on the host, and remap_file_pages is used to update the process page tables to change page mappings underneath the VMA.

Paolo Giarrusso has taken this patch and is making it more acceptable for merging into mainline. This is a challenging process, as the patch is intrusive into some sensitive areas of the VM system. However, the results should be worthwhile, as remap_file_pages produces noticeable performance improvements for UML, and other mmap-intensive applications, such as some databases.

4 Future

So far, I've talked about virtualization enhancements which either already exist or which show

[4]depending on whether the fault was caused by no page being present or the page being mapped with insufficient access for the faulting operation

some promise of existing in the near future. There are a couple of areas where there are problems with no attractive solutions or a solution that needs a good deal of work in order to be possibly mergeable.

4.1 AIO enhancements

4.1.1 Buffered AIO

Currently AIO is only possible in conjunction with `O_DIRECT`. This is where the greatest benefit from AIO is seen. However, there is demand for AIO on buffered data, which is stored in the kernel buffer cache. UML has several filesystems which store data in the host filesystem, and the ability for these filesystems to perform AIO would be welcome. There is a patch to implement this, but it hasn't been merged.

4.1.2 AIO on metadata

Virtual machines would prefer to sleep in the host kernel only when they choose to, and for operations which may sleep to be performed asynchronously and deliver an event of some sort when they complete. AIO accomplishes this nicely for file data. However, operations on file metadata, such as `stat`, can still sleep while the metadata is read from disk. So, the ability to perform `stat` asynchronously would be a nice small addition to the AIO subsystem.

4.1.3 AIO `mmap`

When reading and writing buffered data, it is possible to save memory by mapping the data and modifying the data in memory rather than using `read` and `write`. When mapping a file, there is no copying of the data into the process address space. Rather, the page of data in the kernel's page cache is mapped into the address space.

Against the memory savings, there is the cost of changing the process memory mappings, which can be considerable—comparable to copying a page of data. However, on systems where memory is tight, the option of using `mmap` for guest file I/O rather than `read` and `write` would be welcome.

Currently, there is no support for doing `mmap` asynchronously. It can be simulated (which UML does) by calling `mmap` (which returns after performing the map, but without reading any data into the new page), and then doing an AIO read into the page. When the read finishes, the data is known to be in memory and the page can be accessed with high confidence[5] that the access will not cause a page fault and sleep.

This works well, but real AIO `mmap` support would have the advantage that the cost of the `mmap` and TLB flush could be hidden. If the AIO completes while another process is in context, then the address space of the process requesting the I/O can be updated for free, as a TLB flush would not be necessary.

4.2 Address spaces

UML has a real need for the ability of one process to be able to change mappings within the address space of another. In SKAS (Separate Kernel Address Space) mode, where the UML kernel is in a separate address space from its processes, this is critical, as the UML kernel needs to be able to fix page faults, COW processes address spaces during `fork`, and empty process address spaces during `execve`. In

[5]there is a small chance that the page could be swapped out between the completion of the read and the subsequent access to the data

SKAS3 mode, with the host SKAS patch applied, this is done using a special device which creates address spaces and returns file descriptors that can be used to manipulate them. In SKAS0 mode, which requires no host patches, address space changes are performed by a bit of kernel code which is mapped into the process address space.

Neither of these solutions is satisfactory, nor are any of the alternatives that I know about.

4.2.1 `/proc/mm`

`/proc/mm` is the special device used in SKAS3 mode. When it is opened, it creates a new empty address space and returns a file descriptor referring to it. This address space remains in existence for as long as the file descriptor is open. On the last close, if it is not in use by a process, the address space is freed.

Mappings within a `/proc/mm` address space are changed by writing structures to the corresponding file descriptor. This structure is a tagged union with an arm each for `mmap`, `munmap`, and `mprotect`. In addition, there is a `ptrace` extension, `PTRACE_SWITCH_MM`, which causes the traced child to switch from one address space to another.

From a practical point of view, this has been a great success. It greatly improves UML performance, is widely used, and has been stable on i386 for a long time. However, from a conceptual point of view, it is fatally flawed. The practice of writing a structure to a file descriptor in order to accomplish something is merely an ioctl in disguise. If I had realized this at the time, I would have made it an ioctl. However, the requirement for a new ioctl is usually symptomatic of a design mistake. The use of `write` (or `ioctl`) is an abuse of the interface. It would have been better to implement three new system calls.

4.2.2 New system calls

My proposal, and that of Eric Biederman, who was also thinking about this problem, was to add three new system calls that would be the same as `mmap`, `munmap`, and `mprotect`, except that they would take an extra argument, a file descriptor, which would describe the address space to be operated upon, as shown in Figure 1

This new address space would be returned by a fourth new system call which takes no arguments and returns a file descriptor referring to the address space:

```
int new_mm(void);
```

Linus didn't like this idea, because he didn't want to introduce a bunch of new system calls which are identical to existing ones, except for a new argument. Instead he proposed a new system call which would run any other system call in the context of a different address space.

4.2.3 `mm_indirect`

This new system call is shown in Figure 2.

This would switch to the address space specified by the file descriptor and run the system call described by the second and third arguments.

Initially, I thought this was a fine idea, and I implemented it, but now I have a number of objections to it.

- It is unstructured—there is no type-checking on the system call arguments. This is generally considered undesirable in the system call interface as it makes it impossible for the compiler to detect many errors.

```
int fmmap(int address_space, void *start, size_t length,
          int prot, int flags, int fd, off_t offset);
int fmunmap(int addresss_space, void *start, size_t length);
int fmprotect(int address_space, const void *addr, size_t len,
              int prot);
```

Figure 1: Extended `mmap`, `munmap`, and `mprotect`

```
int mm_indirect(int fd, unsigned long syscall,
                unsigned long *args);
```

Figure 2: `mm_indirect`

- It is too general—it makes sense to invoke relatively few system calls under `mm_indirect`. For UML, I care only about `mmap`, `mprotect`, and `munmap`[6]. The other system calls for which this might make sense are those which take pointers into the process address space as either arguments or output values, but there is currently no demand for executing those in a different address space.
- It has strange corner cases—the implementation of `mm_indirect` has to be careful with address space reference counts. Several system calls change this reference count and `mm_indirect` would need to be aware of these. For example, both `exit` and `execve` dereference the current address space. `mm_indirect` has to take a reference on the new address space for the duration of the system call in order to prevent it disappearing. However, if the indirected system call is exit, it will never return, and that reference will never be dropped. This can be fixed, but the presence of behavior like this suggests that it is a bad idea. Also, the kernel stack could be attacked by nesting `mm_indirect`. The best way to deal with these problems is probably just to disallow running the problematic system calls under `mm_indirect`.
- There are odd implementation problems—for performance reasons, it is desirable not to do an address space switch to the new address space when it's not necessary, which it shouldn't be when changing mappings. However, `mmap` can sleep, and some systems (like SMP x86_64) get very upset when a process sleeps with `current->mm != current->active_mm`.

For these reasons, I now think that `mm_indirect` is really a bad idea.

These are all of the reasonable alternatives that I am aware of, and there are objections to all of them. So, with the exception of having these ideas aired, we have really made no progress on this front in the last few years.

4.3 VCPU

An idea which has independently come up several times is to do virtualization by introduc-

[6]and `modify_ldt` on i386 and x86_64

ing the idea of another context within a process. Currently, processes run in what would be called the privileged context. The idea is to add an unprivileged context which is entered using a new system call. The unprivileged context can't run system calls or receive signals. If it tries to execute a system call or a signal is delivered to it, then the original privileged context is resumed by the "enter unprivileged context" system call returning. The privileged context then decides how to handle the event before resuming the unprivileged context again.

In this scheme, the privileged context would be the UML kernel, and the unprivileged context would be a UML process. This idea has the promise of greatly reducing the overhead of address space switching and system call interception.

In 2004, Ingo Molnar implemented this, but didn't tell anyone until KS 2005. I haven't yet taken a good look at the patch, and it may turn out that it is unneeded given the virtualization infrastructure that is in progress.

5 Conclusion

In the past few years, Linux has greatly improved its ability to host virtual machines. The `ptrace` enhancements have been specifically aimed at virtualization. Other enhancements, such as the I/O changes, have broader application, and were pushed for reasons other than virtualization.

This progress notwithstanding, there are areas where virtualization support could improve. The kernel virtualization infrastructure project holds the promise of greatly reducing the overhead imposed on guests, but these are early days and it remains to be seen how this will play out. If, for some reason, it goes nowhere, Ingo Molnar's VCPU patch is still a possibility.

There are still some unresolved problems, notably manipulating remote address spaces. This aside, all major problems with Linux hosting virtual machines have at least proposed solutions, if they haven't yet been actually solved.

System Firmware Updates Utilizing Sofware Repositories

OR: Two proprietary vendor firmware update packages walk into a dark alley, six RPMS in a yum repository walk out…

Matt Domsch
Dell
Matt_Domsch@dell.com

Michael Brown
Dell
Michael_E_Brown@dell.com

Abstract

Traditionally, hardware vendors don't make it easy to update the firmware (motherboard BIOSes, RAID controller firmware, systems management firmware, etc.) that's flashed into their systems. Most provide DOS-based tools to accomplish this, requiring a reboot into a DOS environment. In addition, some vendors release OS-specific, proprietary tools, in proprietary formats, to accomplish this. Examples include Dell Update Packages for BIOS and firmware, HP Online System Firmware Update Component for Linux, and IBM ServRAID BIOS and Firmware updates for Linux. These tools only work on select operating systems, are large because they carry all necessary prerequisite components in each package, and cannot easily be integrated into existing Linux change management frameworks such as YUM repositories, Debian repositories, Red Hat Network service, or Novell/SuSE YaST Online Update repositories.

We propose a new architecture that utilizes native Linux packaging formats (.rpm, .deb) and native Linux change management frameworks (yum, apt, etc.) for delivering and installing system firmware. This architecture is OS distribution, hardware vendor, device, and change management system agnostic.

The architecture is easy as PIE: splitting Payload, Inventory, and Executable components into separate packages, using package format Requires/Provides language to handle dependencies at a package installation level, and using matching Requires/Provides language to handle runtime dependency resolution and installation ordering.

The framework then provides unifying applications such as `inventory_firmware` and `apply_updates` that handle runtime ordering of inventory, execution, and conflict resolution/notification for all of the plug-ins. These are the commands a system administrator runs. Once all of the separate payload, inventory, and execution packages are in package manager format, and are put into package manager repositories, then standard tools can retreive, install, and execute them:

```
# yum install $(inventory_firmware -b)
# apply_updates
```

We present a proof-of-concept source code implementing the base of this system; web site

and repository containing Dell desktop, notebook, workstation, and server BIOS images; open source tools for flashing Dell BIOSes; and open source tools to build such a repository yourself.

1 Overview

The purpose of this paper is to describe a proposal and sample implementation to perform generic, vendor-neutral, firmware updates using a system that integrates cleanly into a normal Linux environment. Firmware includes things such as system BIOS; addon-card firmware, e.g. RAID cards; system Baseboard Management (BMC), hard drives, etc. The first concept of the proposal is the definition of a basic update framework and a plugin API to inventory and update the system. For this, we define some basic utilities upon which to base the update system. We also define a plugin architecture and API so that different vendor tools can cleanly integrate into the system. The second critical piece of the update system is cleanly separating system inventory, execution of updates, and payload, i.e. individual firmware images. After defining the basic utilities to glue these functions together, we define how a package management system should package each function. Last, we define the interaction between the package manager and the repository manager to create a defined interface for searching a repository for applicable updates. This paper will cover each of these points. The proposal describes an implementation, called `firmware-tools` [1].

2 Infrastructure

This section will detail the basic components used by the firmware update system, and firmware-tools. The base infrastructure for this system consists of two components: inventory and execution. These are named `inventory_firmware` and `apply_updates`, respectively. These are currently command-line utilities, but it is anticipated that, after the major architectural issues are worked out and this has been more widely peer-reviewed, there will be GUI wrappers written.

The basic assumption is that, before you can update firmware on a device, you need several pieces of information.

- What is the existing firmware version?
- What are the available versions of firmware that are on-disk?
- How do you do a version comparison?
- How do I get the correct packages installed for the hardware I have? In other words, solve the bootstrap issue.

It is important to note that all of these questions are independent of exactly how the files and utilities get installed on the system. We have deliberately split out the behavior of the installed utilities from the specification of how these utilities are installed. This allows us flexibility in packaging the tools using the "best" method for the system. Packaging will be discussed in a section below. The specification of packaging and how it interacts with the repository layer is an important aspect of how the initial set of utilities get bootstrapped onto the system, as well as how payload upgrades are handled over time.

2.1 Existing Firmware Version

The answer to the question, "What is the existing firmware version?" is provided by the inventory_firmware tool. The basic inventory_firmware tool has no capability to inventory anything; all inventory capability is provided by plugins. Plugins consist of a python module with a specific entry point, plus a configuration fragment to tell inventory_firmware about the plugin. Each plugin provides inventory capability for one device type. The plugin API is covered in the API section of this paper, below. It should be noted that, at this point, the plugin API is still open for suggestions and updates.

As an example, there is a dell-lsiflash package that provides a plugin to inventory firmware on LSI RAID adapters. The dell-lsiflash plugin package drops a configuration file fragment into the plugin directory /etc/firmware/firmware.d/ in order to activate the plugin. This configuration file fragment looks like this:

```
[delllsi]
# plugin that provides
# inventory for LSI RAID cards.
inventory_plugin=delllsi
```

This causes the inventory_plugin to load a python module named delllsi.py and use the entry points defined there to perform inventory on LSI RAID cards. The delllsi.py module is free to do the inventory any way it chooses. For example, there are vendor utilities that can sometimes be re-purposed to provide quick and easy inventory. In this specific case, we have written a small python extension module in C which calls a specific ioctl() in the LSI megaraid drivers to perform the inventory and works across all LSI hardware supported by the megaraid driver family. Note that while the framework is open source, the per-device inventory applications may choose their own licenses (of course, open source apps are strongly preferred).

2.2 Available Firmware Images

The next critical part of infrastructure lies in enumerating the payload files that are available on-disk. The main firmware-tools configuration file defines the top-level directory where firmware payloads are stored. The default location for firmware images is /usr/share/firmware/. This can be changed such that, for example, multiple systems network mount a central repository of firmware images. In general each type or class of firmware update will create a subdirectory under the main top-level directory, and each individual firmware payload will have another subdirectory under that.

Each individual firmware payload consists of two files: a binary data file of the firmware and a package.ini metatdata file used by the firmware-tools utilities. It specifies the modules to be used to apply the update and the version of the update, among other things.

2.3 Version Comparison

Another interesting problem lies in doing version comparison between different version strings to try to figure out which is newer, due to the multitude of version string formats used by different firmware types. For example, some firmware might have version strings such as A01, A02, etc., while other firmware has version strings such as 2.7.0-1234, 2.8.1-1532, etc. Each different system may have different precedence rules. For example, current Dell BIOS releases have version strings in sequence like A01, A02, etc. But non-release, beta BIOS have version strings like X01, X02, etc., and developer test BIOS have version strings like P01, P02, etc. This poses a problem because a naive string comparison would always rank beta "X-rev" BIOS as higher version than production BIOS, which is undesirable.

The solution to this problem is to allow plugins to define version comparison functions. These functions take two strings as input and output which one is newer. Each `package.ini` configuration file contains the payload version, plus the name of the plugin to use for version comparison.

2.4 Initial Package Installation—Bootstrap

The last interesting problem arises when you consider how to decide which packages to download from the package repository and install on the local machine. This is a critical problem to solve in order to drive usability of this solution. If the user has to know details of the machine to manually decide which packages to download, then the system will not be sucessful. Next to consider is that a centralized solution does not fit in well with the distributed nature of Linux, Linux development, and the many vendors we hope to support with this solution. We aim to provide a distributed solution where the packages themselves carry the necessary metadata such that a repository manager metadata query can provide an accurate list of which package is needed.

Normal package metadata relates to the software in the package, including files, libraries, virtual package names, etc. The firmware-tools concept extends this by defining "applicability metadata" and adding it to the payload packages. For example, we add
`Provides: pci_firmware(...)`
RPM tags to tell that the given RPM file is applicable to certain PCI cards. Details on packaging are in the next section, including specifications on package Provides that must be in each package.

We then provide a special "bootstrap inventory" mode for the inventory tool. In this mode, `inventory_firmware` outputs a standardized set of package Provides names, based upon the current system hardware configuration. By default, this list only includes `pci_firmware(...)`. Additional vendor-specific addon packs can add other, vendor-specific package names. For example, the Dell addon pack, firmware-addon-dell, adds `system_bios(...)` and `bmc_firmware(...)` standard packages to the list. We hope for wide vendor adoption in this area, where different vendors can provide addon packs for their standard systems. In this manner, the user need not know anything about their hardware, other than the manufacturer. They simply ask their repository manager to install the addon pack for their system. They then run bootstrap inventory to get a list of all other required packages. This list is fed to the OS repository manager, for example, yum, up2date, apt, etc. The repository manager will then search the repository for packages with matching Provides names. This package will normally be the firmware payload package. Through the use of Requires, the payload packages will then pull the execution and inventory packages into the transaction.

3 plugin-api

The current `firmware-tools` provides only infrastructure. All actual work is done by writing plugins to do either inventory, bootstrap, or execution tasks. We expect that as new members join the firmware-tools project this API will evolve. The current API is very straightforward, consisting of a configuration file, two mandatory function calls, and one optional function call. It is implemented in python, but we anticipate that in the future we may add a C API, or something like a WBEM API. The strength of the current implementation is its simplicity.

3.1 Configuration

Plugins are expected to write a configuration file fragment into `/etc/firmware/firmware.d/`. This fragment should be named `modulename.conf`. It is an `INI`-format configuration file that is read with the python ConfigParser module. Each configuration fragment should have one section named the same as the plugin, for example, `[delllsi]`. At the moment, there are only two configuration directives that can be placed in this section. The first is, `bootstrap_inventory_plugin=` and the other is `inventory_plugin=`.

3.2 Bootstrap Inventory

When in bootstrap mode, `inventory_firmware` searches the configuration for `bootstrap_inventory_plugin=` directives. It then dynamically loads the specified python module. It then calls the `BootstrapGenerator()` function in that module. This function takes no arguments and is expected to be a python "generator" function [2]. This function yields, one-by-one, instances of the `package.InstalledPackage` class.

Figure 1 illustrates the Dell bootstrap generator for the `firmware-addon-dell` package.

This module is responsible for generating a list of all possible packages that could be applicable to Dell systems. As you can see, it outputs two standard packages, `system_bios(...)` and `bmc_firmware(...)`. It is also responsible for outputting a list of `pci_firmware(...)` packages with the system name appended. In the future, as more packages are added to the system, we anticipate that the bootstrap will also output package names for things such as external SCSI/SAS enclosures, system backplanes, etc.

3.3 System Inventory

When in system inventory mode, `inventory_firmware` searches the configuration for `inventory_plugin=` directives. It then dynamically loads the specified python module. It then calls the `InventoryGenerator()` function in that module. This function takes no arguments and is expected to be a python "generator" function. This function yields, one-by-one, instances of the `package.InstalledPackage` class. The difference here between this and bootstrap mode is that, in system inventory mode, the inventory function will populate `version` and `compareStrategy` fields of the `package.InstalledPackage` class.

Figure 2 illustrates the Dell inventory generator for the firmware-addon-dell package.

The inventory generator in this instance outputs only the BIOS inventory, with more detailed version information. It is also responsible for setting up the correct comparison function to use for version comparison purposes.

3.4 On-Disk Payload Repository

The on-disk payload repository is the toplevel directory where firmware payloads are stored. There is currently not a separate tools to generate an inventory of the repository, but, there is python module code in `repository.py` which will provide a list of available packages in the on-disk repository. The `repository.Repository` class handles the on-disk repository. The constructor should be given the top-level directory. After construction, the `iterPackages()` or `iterLatestPackages()` generator function methods can be called to get a list of packages in the repository. These generator functions output either all repository packages,

```
# standard entry point -- Bootstrap
def BootstrapGenerator():
  # standard function call to get Dell System ID
  sysId = biosHdr.getSystemId()

  # output packages for Dell BIOS and BMC
  for i in [ "system_bios(ven_0x1028_dev_0x%04x)", "bmc_firmware(ven_0x1028_dev_0x%04x)" ]:
    p = package.InstalledPackage(
        name = (i % sysId).lower()
        )
    yield p

  # output all normal PCI bootstrap packages with system-specific name appended.
  module = __import__("bootstrap_pci", globals(),  locals(), [])
  for pkg in module.BootstrapGenerator():
    pkg.name = "%s/%s" % (pkg.name, "system(ven_0x1028_dev_0x%04x)" % sysId)
    yield pkg
```

Figure 1: Dell bootstrap generator code

```
# standard entry point -- Inventory
def InventoryGenerator():
  sysId = biosHdr.getSystemId()
  biosVer = biosHdr.getSystemBiosVer()
  p = package.InstalledPackage(
    name = ("system_bios(ven_0x1028_dev_0x%04x)" % sysId).lower(),
    version = biosVer,
    compareStrategy = biosHdr.compareVersions,
    )
  yield p
```

Figure 2: Dell inventory generator code

or only latest packages, respectively. They read the `package.ini` file for each package and output an instance of `package.RepostitoryPackage`. The `package.ini` specifies the wrapper to use for each repository package object. The wrapper will override the `compareVersion()` and `install()` methods as appropriate.

3.5 Execution

Execution is handled by calling the `install()` on a package object returned from the repository inventory. The `install()` method is set up by a type-specific wrapper, as specified in the `package.ini` file. Figure 3 shows a typical wrapper class.

The wrapper constructor is passed a package object. The wrapper will then set up methods in the package object for install and version compare. Typical installation function is a simple call to a vendor command line tool. In this example, it uses the open-source `dell_rbu` kernel driver and the open-source libsmbios [3] `dellBiosUpdate` application to perform the update.

4 Packaging

The goal of packaging is to make it as easy as possible to integrate firmware update applications and payloads into existing OS deployments. This means following a standards-based packaging format. For Linux, this is the Linux Standard Base-specified Red Hat Package Manager (RPM) format, though we don't preclude native Debian or Gentoo package formats. The concepts are equally applicable; implementation is left as an exercise for the reader.

Base infrastructure components are in the `firmware-tools` package, detailed previously. Individual updates for specific device classes are split into two (or more) packages: an Inventory and Execution package, and a Payload package. The goal is to be able to provide newer payloads (the data being written into the flash memory parts) separate from providing newer inventory and execution components. In an ideal world, once you get the relatively simple inventory and execution components right, they would rarely have to change. However, one would expect the payloads to change regularly to add features and fix bugs in the product itself.

4.1 RPM Dependencies

Payload packages have a one-way (optionally versioned) RPM dependency on the related Inventory and Execution package. This allows tools to request the payload package, and the related Inventory and Execution package is downloaded as well. Should there be a compelling reason to do so, the Inventory and Execution components may be packaged separately, though most often they're done by the same tool.

Payload packages further Provide various tags, again to simplify automated download tools.

Lets look at the details, using BIOS package for Dell PowerEdge 6850 as an example. The actual BIOS firmware image is packaged in an RPM called `system_bios_PE6850-a02-12.3.noarch.rpm`. This package has RPM version-release `a02-12.3`, and is a `noarch` rpm because it does not contain any CPU architecture-specific executable content.

This package Provides:

```
class BiosPackageWrapper(object):
  def __init__(self, package):
    package.installFunction = self.installFunction
    package.compareStrategy = biosHdr.compareVersions
    package.type = self

  def installFunction(self, package):
    ret = os.system("/sbin/modprobe dell_rbu")
    if ret:
        out = ("Could not load Dell RBU kernel driver (dell_rbu).\n"
              " This kernel driver is included in Linux kernel 2.6.14 and later.\n"
              " For earlier releases, you can download the dell_rbu dkms module.\n\n"
              " Cannot continue, exiting...\n")
        return (0, out)
    status, output = commands.getstatusoutput("""dellBiosUpdate -u -f %s""" %
                     os.path.join(package.path, "bios.hdr"))
    if status:
        raise package.InstallError(output)
    return 1
```

Figure 3: Example wrapper class

```
system_bios(ven_0x1028
_dev_0x0170) = a02-12.3
system_bios_PE6850 = a02-12.3
```

Let's look at these one at a time.

```
system_bios(ven_0x1028
_dev_0x0170) = a02-12.3
```

This can be parsed as denoting a system BIOS, from a vendor with PCI SIG Vendor ID number of 0x1028 (Dell). For each vendor, there will be a vendor-specific system type numbering scheme which we care nothing about except to consume. In this example, 0x0170 is the software ID number of the PowerEdge 6850 server type. The BIOS version, again using a vendor-specific versioning scheme, is A02. All of the data in these fields can be determined programatically, so is suitable for automated tools.

Most systems and devices will have prettier, marketing names. Whenever possible, we want to use those, rather than the ID numbers, when interacting with the sysadmin. So this package also provides the same version information, only now using the marketing short name `PE6850`.

```
system_bios_PE6850 = a02-12.3
```

Presumably the marketing short names, though per-vendor, will not conflict in this flat namespace. The BIOS version, A02, is seen here again, as well as a release field (12.3) which can be used to indicate the version of the various tools used to produce this payload package. This version-release value matches that of the RPM package.

The firmware-addon-dell package provides an ID-to-shortname mapping config appropriate for Dell-branded systems. It is anticipated that other vendors will provide equivalent functionality for their packages. Users generating their own content for systems not in the list can accept the auto-generated name or add their system ID to the mapping config.

Epochs are used to account for version scheme changes, such as Dell's conversion from the Axx format to the `x.y.z` format.

To account for various types of firmware that may be present on the system, we have come up with a list for RPM Provides tags seen in Figure 4. We anticipate adding new entries to this list as firmware updates for new types of devices are added to the system.

The combination pci_firmware/system entries

```
system_bios(ven_VEN_dev_ID)
pci_firmware(ven_VEN_dev_DEV)
pci_firmware(ven_VEN_dev_DEV_subven_SUBVEN_subdev_SUBDEV)
pci_firmware(ven_VEN_dev_DEV_subven_SUBVEN_subdev_SUBDEV)/system(ven_VEN_dev_ID)
bmc_firmware(ven_VEN_dev_ID)

system_bios_SHORTNAME
pci_firmware_SHORTNAME
pci_firmware_SHORTNAME/system_SHORTNAME
bmc_firmware_SHORTNAME
```

Figure 4: Package Manager Provides lines in payload packages

are to address strange cases where a given payload is applicable to a given device in a given system only, where the PCI `ven/dev/subven/subdev` values aren't enough to disambiguate this. It's very rare, and should be used with extreme caution, if at all.

These can be expanded to add additional firmware types, such as SCSI backplanes, hot plug power supply backplanes, disks, etc. as the need arises. These names were chosen to avoid conflicts with existing RPM packages Provides.

4.2 Payload Package Contents

Continuing our BIOS example, the toplevel firmware storage directory is `/usr/share/firmware`. BIOS has its own subdirectory under the toplevel, at `/usr/share/firmware/bios/`, representing the top-level BIOS directory. The BIOS RPM payload packages install their files into subdirectories of the BIOS toplevel directory. Figure 5 shows this layout.

This allows multiple versions of each payload to be present on the file system, which may be handy for downrev'ing. It also allows an entire set of packages to be installed once on a file server and shared out to client servers.

In this example, the actual data being written to the flash is in the file `bios.hdr`. The `package.ini` file contains metadata about the payload described above and consumed by the framework apps. The `package.xml` file listed here was copied from the original vendor package. It contains additional metadata, and may be used by vendor-specific tools. The `firmware-addon-dell` package uses the information in this file to only attempt installing the payload onto the system type for which it was made (e.g. to avoid trying to flash a desktop system with a server BIOS image).

4.3 Obtaining Payload Content

We've described the format of the packages, but what if the existing update tools aren't already in the proper format? For example, as detailed at the beginning of this paper, most vendors release their content in proprietary formats. The solution is to write a tool that will take the existing proprietary formats and repackage them into the `firmware-tools` format.

The `fwupdate-tools` package provides a script, `mkbiosrepo.sh`, which can download files from `support.dell.com`, extract and unpack the relevant payloads from them, and re-package them into packages as we've described here. This allows a graceful transition from an existing packaging format to this new format with little impact to existing business processes. The script can be extended to do likewise for other proprietary vendor package formats.

```
# rpm -qpl system_bios_PE6850-a02-12.3.noarch.rpm
/usr/share/firmware/bios
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02/bios.hdr
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02/package.ini
/usr/share/firmware/bios/system_bios_ven_0x1028_dev_0x0170_version_a02/package.xml
```

Figure 5: Example Package Manager file layout

If this format proves to be popular, it is hoped that vendors will start to release packages in native firmware-tools format. The authors of this paper are already working internally to Dell to push for this change, although there is currently no ETA nor guarantee of official Dell support. We are working on the open-source firmware-tools project to prototype the solution and to get peer review on this concept from other industry experts in this area.

5 Repositories

We recognize that each OS distribution has its own model for making packages avaialble in an online repository. Red Hat Enterprise Linux customers use Red Hat Network, or RHN Satellite Server, to host packages. Fedora and CentOS use Yellow dog Updater, Modified (YUM) repositories. SuSE uses Novell ZenWorks, YaST Online Update (YOU) repositories, and newer SuSE releases can use YUM repositories too. Debian uses FTP archives. Other third party package managers have their own systems and tools. The list goes on and on. In general, you can put RPMs or debs into any of these, and they "just work."

As an optimization, you can package RPMs in a single directory, and provide the multiple forms of metadata that each require in that same location, letting one set of packages, and one repository, be easily used by all of the system types. The `mkbiosrepo.sh` script manages metadata for both YUM and YOU tools. Creation of channels in Red Hat Network Satellite Server is, unfortunately, a manual process at present; uploading content into channels is easily done using RHN tools. Providing packages in other repository formats is another exercise left to the reader.

6 System Administrator Use

Up to this point, everything has focused on creating and publishing packages in a format for system administration tools to consume. So how does this all look from the sysadmin perspective?

6.1 Pulling from a Repository

First, you must configure your target systems to be able to pull files from the online repositories. How you do that is update system specific, but it probably involves editing a configuration file (`/etc/yum.repos.d/`, `/usr/sysconfig/rhn/sources`, ...) to point at the repository, configure GPG keys, and the like. Nothing here is specific to updating firmware.

The first tool you need is one that will match your system vendor, which pulls in the framework packages, which provides the `inventory_firmware` tool.

```
# yum install firmware-addon-dell
```

6.2 Bootstrapping from a Repository

Now it's time to request from the repository all the packages that might match your target system. `inventory_firmware`, in bootstrap mode, provides the list of packages that could exist. Figure 6 shows an example.

We pass this value to yum or up2date, as such:

```
# yum install $(inventory_firmware
-b)
```

or

```
# up2date -i $(inventory_firmware
-b -u)
```

This causes each of the possible firmware Payload packages, if they exist in any of the repositories we have configured to use, to be retreived and installed into the local file system. Because the Payload packages have RPM dependencies on their Inventory and Execution packages, those are downloaded and installed also.

Subsequent update runs, such as the nightly yum or up2date run will then pick up any newer packages, using the list of packages actually on our target system. If packages for new device types are released into the repository (e.g. someone adds disk firmware update capability), then the sysadmin will have to run the above commands again to download those new packages.

6.3 Applying Firmware Updates

`apply_updates` will perform the actual flash part update using the inventory and execution tools and payloads for each respective device type.

```
# apply_updates
```

`apply_updates` can be configured to run automatically at RPM package installation time, though its more likely to be run as a scheduled downtime activity.

7 Proof of Concept Payload Repository

Using the above tool set, we've created a proof-of-concept payload repository [4], containing the latest Dell system BIOS for over 200 system types, and containing Dell PERC RAID controller firmware for current generation controllers. It provides YUM and YOU metadata in support of target systems running Fedora Core 3, 4, and 5, Red Hat Enterprise Linux 3 and 4 (and its clones like CentOS), and Novell/SuSE Linux Enterprise Server 9 and 10. New device types and distributions will be added in the future.

8 Future Directions

We believe that this model for automatically downloading firmware can also be used for other purposes. For example, we could tag DKMS [5] driver RPMS with tags and have the inventory system output `pci_driver(...)` lines to be fed into yum or up2date. A proposal has been sent to the dkms mailing list with subsequent commentary and discussion. This model could also be used for things like Intel ipw2x00 firmware, which typically is downloaded separately from the kernel and must match the kernel driver version.

9 Conclusion

While most sysadmins only update their BIOS and firmware when they have to, the process

```
# inventory\_firmware -b
system_bios(ven_0x1028_dev_0x0170)
bmc_firmware(ven_0x1028_dev_0x0170)
pci_firmware(ven_0x8086_dev_0x3595)/system(ven_0x1028_dev_0x0170)
pci_firmware(ven_0x8086_dev_0x3596)/system(ven_0x1028_dev_0x0170)
pci_firmware(ven_0x8086_dev_0x3597)/system(ven_0x1028_dev_0x0170)
...
```

Figure 6: Running `inventory_firmware -b`

should be as easy as possible. By utilizing OS tools already present, BIOS and firmware change management becomes just as easy as other software change management. We've developed this to be Linux distribution, hardware manufacturer, system manufacturer, and update mechanism agnostic, and have demonstrated its capability with Dell BIOS and PERC Firmware on a number of Linux distributions and versions. We encourage additional expansion of the types of devices handled, types of OSs, and types of update systems, and would welcome patches that provide this functionality.

10 Glossary

Package: OS standard package (`.rpm`/`.deb`)

Package Manager: OS standard package manager (`rpm`/`dpkg`)

Repository Manager: OS standard repository solution (`yum`/`apt`)

References

[1] Firmware-tools Project
Home page: `http://linux.dell.com/firmware-tools/`
Mailing list: `http://lists.us.dell.com/mailman/listinfo/firmware-tools-devel`

[2] Python Generator documentation
`http://www.python.org/dev/peps/pep-0255/`

[3] Libsmbios Project
Home Page: `http://linux.dell.com/libsmbios`
Mailing list: `http://lists.us.dell.com/mailman/listinfo/libsmbios-devel`

[4] Proof of Concept Payload Repository
Home Page: `http://fwupdate.com`

[5] DKMS Project
Home Page:
`http://linux.dell.com/dkms`
Mailing list:
`http://lists.us.dell.com/mailman/listinfo/dkms-devel`

The Need for Asynchronous, Zero-Copy Network I/O

Problems and Possible Solutions

Ulrich Drepper
Red Hat, Inc.
`drepper@redhat.com`

Abstract

The network interfaces provided by today's OSes severely limit the efficiency of network programs. The kernel copies the data coming in from network interface at least once internally before making the data available in the user-level buffer. This article explains the problems and introduces some possible solutions. These necessarily cover more than just the network interfaces themselves, there is a bit more support needed.

1 Introduction

Writing scalable network applications is today more challenging than ever. The problem is the antiquated (standardized) network API. The Unix socket API is flexible and remains usable but with ever higher network speeds, new technologies like interconnects, and the resulting expected scalability we reach the limits. CPUs and especially their interface to the memory subsystem are not capable of dealing with the high volume of data in the available short time-frames.

What is needed is an asynchronous interface for networking. The asynchronicity would primarily be a means to avoid unnecessary copying of data. It also would help to avoid congestion since network buffers can earlier be freed which in turn ensures that retransmits due to full network buffers are minimized.

Existing interfaces like the POSIX AIO functions fall short of providing the necessary functionality. This is not only due to the fact that pre-posting of buffers is only possible at a limited scale. A perhaps bigger problem is the expensive event handling. The event handling itself has requirements and challenges which currently cannot be worked around (like waking up too many waiters).

In the remainder of the paper we will see the different set of interfaces which are needed:

- event handling
- physical memory handling
- asynchronous network interfaces

The event handling must work with the existing `select()`/`poll()` interfaces. It also should be generic enough to be usable for other events which currently do not map to file descriptors in the moment (like message queues, futexes, etc). This way we might *finally* have one unified inner loop in the event handling of a program.

Physical memory suddenly becomes important because network devices address only physical memory. But physical memory is invisible in the Unix ABI and this is very much welcome. If this is not the case the Copy-On-Write concept used on CPUs with MMU-support would not work. The implementation of functions like `fork()` becomes harder. The proposal will center around making physical memory regions objects the kernel knows to handle.

Finally, using event and physical memory handling, it is possible to define sane interfaces for asynchronous network handling. In the following sections we will see some ideas on how this can happen.

It is not at all guaranteed that these interfaces will stand the test of time or will even be implemented. The intention of this paper is to get the ball rolling because the problems are pressing and we definitely need something along these lines. Starting only from the bottom (i.e., from the kernel implementation) has the danger of ignoring the needs of programmers and might miss the bigger picture (e.g., integration into a bigger event handling scheme, for instance).

2 The Existing Implementation

Network stacks in Unix-like OSes have more or less the same architecture today as they had 10–20 years ago. The interface the OS provides for reading and writing from and to network interfaces consists of the interfaces in Table 1.

These interfaces all work synchronously. The interfaces for reading return only when data is available or in error conditions. In non-blocking mode they can return immediately if no data is available, but this is no real asynchronous handling. The data is not transferred to userlevel in an asynchronous fashion.

receiving	sending
`read()`	`write()`
`recv()`	`send()`
`recvfrom()`	`sendto()`
`recvmsg()`	`sendmsg()`

Table 1: Network APIs

receiving	sending
`read()`[1]	`write()`[1]
`aio_read()`	`aio_write()`
`lio_listio()`	

Table 2: AIO APIs

Linux provides an asynchronous mode for terminals, sockets, pipes, and FIFOs. If a file descriptor has the `O_ASYNC` flag set calls to `read()` and `write()` immediately return and the kernel notifies the program about completion by sending a signal. This is a slightly more complicated and more restrictive version of the AIO interfaces than when using `SIGEV_SIGNAL` (see below) and therefore suffers in addition to its own limitation of those of `SIGEV_SIGNAL`.

For truly asynchronous operations on files the POSIX AIO functions from Table 2 are available. With these interfaces it is possible to submit a number of input and output requests on one or more file descriptors. Requests are filled as the data becomes available. No particular order is guaranteed but requests can have priorities associated with them and the implementation is supposed to order the requests by priority. The interfaces also have a synchronous mode which comes in handy from time to time. Interesting here is the asynchronous mode. The big problem to solve is that somehow the program has to be able to find out when the submitted requests are handled. There are three modes

[1]With the `O_ASYNC` flag set for the descriptor.

defined by POSIX:

SIGEV_SIGNAL The completion is signaled by sending a specified signal to the process. Which thread receives the signal is determined by the kernel by looking at the signal masks. This makes it next to impossible to use this mechanism (and `O_ASYNC`) in a library which might be linked into arbitrary code.

SIGEV_THREAD The completion is signaled by creating a thread which executes a specified function. This is quite expensive in spite of NPTL.

SIGEV_NONE No notification is sent. The program can query the state of the request using the `aio_error()` interface which returns `EINPROGRESS` in case the request has not yet been finished. This is also possible for the other two modes but it is crucial for `SIGEV_NONE`.

The POSIX AIO interfaces are designed for file operations. Descriptors for sockets might be used with the Linux implementation but this is not what the functions are designed for and there might be problems.

All I/O interfaces have some problems in common: the caller provides the buffer into which the received data is stored. This is a problem in most situation for even the best theoretical implementation. Network traffic arrives asynchronously, mostly beyond the control of the program. The incoming data has to be stored somewhere or it gets lost.

To avoid copying the data more than once it would therefore be necessary to have buffers usable by the user available right the moment when the data arrives. This means:

- for the `read()` and `recv()` interfaces it would be necessary that the program is making such a call just before when the data arrives. If there is no such call outstanding the kernel has to use its own buffers or (for unreliable protocols) it can discard the data.
- with `aio_read()` and the equivalent `lio_listio()` operation it is possible to pre-post a number of buffers. When the number goes down more buffers can be pre-posted. The main problem with these interfaces is what happens next. Somehow the program needs to be notified about arrival of data. The three mechanisms described above are either based on polling (`SIGEV_NONE`) or are far too heavy-weight. Imagine sending 1000s of signals a second corresponding to the number of incoming packages. Creating threads is even more expensive.

Another problem is that for unreliable protocols it might be more important to always receive the last arriving data. It might contain more relevant information. In this case data which arrived before should be sacrificed.

A second problem all implementations have in common is that the caller can provide arbitrary memory regions for input and output buffer to the kernel. This is in general wanted. But if the network hardware is supposed to transfer directly into the memory regions specified it is necessary for the program to use memory that is special. The network hardware uses Direct Memory Access (DMA) to write into RAM instead of passing data through the CPU. This happens at a level below the virtual address space management, DMA only uses physical addresses.

Besides possible limitations on where the RAM for the buffers is located in the physical address space, the biggest problem is that the buffers must remain in RAM until used. Ordinarily userlevel programs do not see physical RAM; the virtual address is an abstraction and the OS might decide to remove memory pages from RAM to make room for other processes. If this would appear while an network I/O request is pending the DMA access of the network hardware would touch RAM which is now used for something else.

This means while buffers are used for DMA they must not be evicted from RAM. They must be locked. This is possible with the `mlock()` interface but this is a privileged operation. If a process would be able to lock down arbitrary amounts of memory it would impact all the other processes on the system which would be starved of resources. Recent Linux kernels allow unprivileged processes to lock down a modest amount of memory (by default eight pages or so) but this would not be enough for heavily network oriented applications.

The POSIX AIO interfaces certainly show the way for the interfaces which can solve the networking problems. But we have to solve several problems:

- make DMA-ready memory available to unprivileged applications;
- create an efficient event handling mechanism which can handle high volumes of events;
- create I/O interfaces which can use the new memory and event handling. As a bonus they should be usable for disk I/O as well.

At this point it should be mentioned that a working group of the OpenGroup, the Interconnect Software Consortium, tried to tackle this problem. The specification is available from their website at `http://www.opengroup.org/icsc/`. They arrived at the same set of three problems and proposed solutions. Their solutions are not implemented, though, and they have some problems. Most importantly, the event handling does not integrate with the file-descriptor-based event handling.

3 Memory Handling

The main requirement on the memory handling is to provide memory regions which are available at userlevel and which can be directly accessed by hardware other than the processor. Network cards and disk controllers can transfer data without the help of the CPU through DMA. DMA addresses memory based on the physical addresses. It does not matter how the physical memory is currently used. If the virtual memory system of the OS decides that a page of RAM should be used for some other purpose the devices would overwrite the new user's memory unless this is actively prevented. There is no demand-paging as for the userlevel code.

To be sure the DMA access will use the correct buffer, it is necessary to prevent swapping the destination pages out. This is achieved by using `mlock()`. Memory locking depletes the amount of RAM the system can use to keep as much of the combined virtual memory of all processes in RAM. This can severely limit the performance of the system or eventually prevent it from making any progress. Memory locking is therefore a privileged operation. This is the first problem to be solved.

The situation is made worse by the fact that locking can only be implemented on a per-page-basis. Locking one small object on a page ties down the entire page.

One possibility would be avoid locking pages in the program and have the kernel instead do the work all by itself and on-demand. That means if a network I/O request specifies a buffer the kernel could automatically make sure that the memory page(s) containing the buffer is locked. This would be the most elegant solution from the userlevel point-of-view. But it would mean significant overhead: for every operation the memory page status would have to be checked and if necessary modified. Network operations can be frequent and multiple buffers can be located on the same page. If this is known the checks performed by the kernel would be unnecessary and if they are performed the kernel must keep track how many DMA buffers are located on the page. This solution is likely to be unattractive.

It is possible to defer solving this problem, fully or in part, to the user. In the least accommodating solution, the kernel could simply require the userlevel code to use `mmap()` and `mprotect()` with a new flag to create DMA-able memory regions. Inside these memory regions the program can carve out individual buffers, thereby mitigating the problem of locking down many pages which are only partially used as buffers. This solution puts all the burden on the userlevel runtime.

It also has a major disadvantage. Pages locked using `mlock()` are locked until they are unlocked or unmapped. But for the purpose of DMA the pages need not be permanently locked. The locking is really only needed while I/O requests using DMA are being executed. For the network I/O interfaces we are talking about here the kernel always knows when such a request is pending. Therefore it is theoretically possible for the kernel to lock the pages on request. For this the pages would have to be specially marked. While no request is pending or if a network interface is used which does not provide DMA access the virtual memory subsystem of the OS can move the page around in physical memory or even swap it out.

One relatively minor change to the kernel could allow for such optimizations. If the `mmap()` call could be passed a new flag `MAP_DMA` the kernel would know what the buffer is used for. It could keep track of the users of the page and avoid locking it unless it is necessary. In an initial implementation the flag could be treated as an implicit `mlock()` call. If the flag is correctly implemented it would also be possible to specify different limits on the amount of memory which can be locked and for DMA respectively. This is no full solution to the problem of requiring privileges to lock memory, though (an application could simply have a `read()` call pending all the time).

The `MAP_DMA` flag could also help dealing with the effects of `fork()`. The POSIX specification requires that no memory locking is inherited by the child. File descriptors are inherited on the other hand. If parts of the solution for the new network interfaces uses file descriptors (as it is proposed later) we would run into a problem: the interface is usable but before the first use it would be necessary to re-lock the memory. With the `MAP_DMA` flag this could be avoided. The memory would simply automatically re-locked when it is used in the child for the first time. To help in situations where the memory is not used at all after `fork()`, for example, if an `exec` call immediately follows, all `MAP_DMA` memory is unlocked in the child.

Using this one flag alone could limit the performance of the system, though. The kernel will always have to make sure that the memory is locked when an I/O request is pending. This is overhead which could potentially be a limiting factor. The programmer oftentimes has better knowledge of the program semantics. She would know which memory regions are used for longer periods of time so that one explicit lock might be more appropriate than implicit locking performed by the kernel.

A second problem is fragmentation. A program is usually not one homogeneous body of code. Many separate libraries are used which all could perform network I/O. With the `MAP_DMA` method proposed so far each of the libraries would have to allocate its own memory region. This use of memory might be inefficient because of the granularity of memory locking and because not all parts of the program might need the memory concurrently.

To solve the issue, the problem has to be tackled at a higher level. We need to abstract the memory handling. Providing interfaces to allocate and deallocate memory would give the implementation sufficient flexibility to solve these issues and more. The allocation interfaces could still be implemented using the `MAP_DMA` flag and the allocation functions could “simply” be userlevel interfaces and no system calls. One possible set of interfaces could look like this:

```
int dma_alloc(dma_mem_t *handlep,
  size_t size, unsigned int flags);
int dma_free(dma_mem_t handle,
  size_t size);
```

The interfaces which require DMA-able memory would be passed a value of type `dma_mem_t`. How this handle is implemented would be implementation defined and could in fact change over time. An initial, trivial implementation could even do without support for something like `MAP_DMA` and use explicit `mlock()` calls.

`epoll_wait()`	`poll()`	`select()`
`epoll_pwait()`	`ppoll()`	`pselect()`

Table 3: Notification APIs

4 Event Handling

The existing event handling mechanisms of POSIX AIO uses polling, signals, or the creation of threads. Polling is not a general solution. Signals are not only costly, they are also unreliable. Only a limited, small number of signals can be outstanding at any time. Once the limit is reached a program has to fall back on alternative mechanisms (like polling) until the situation is rectified. Also, due to the limitations imposed on code usable in signal handlers, writing programs using signal notification is awkward and error-prone. The creation of threads is even more expensive and despite the speed of NPTL has absolute no chance to scale with high numbers of events.

What is needed is a completely new mechanism for event notification. We cannot use the same mechanisms as used for synchronous operations on a descriptor for a socket or a file. If data is available and can be sent, this does not mean that an asynchronously posted request has been fulfilled.

The structure of a program designed to run on a Unix-y system requires that the event mechanism can be used with the same interfaces used today for synchronous notification (see Table 3). It would be possible to invent a completely new notification handling mechanism and map the synchronous file descriptor operation to it. But why? The existing mechanism work nicely, they scale well, and programmers are familiar with them. It also means existing code does not have to be completely rewritten.

Creating a separate channel (e.g., file descriptor) for each asynchronous I/O request is not scalable. The number of I/O requests can be high enough to forbid the use of the `poll` and `select` interfaces. The `epoll` interfaces would also be problematic because for each request the file descriptor would have to be registered and later unregistered. This overhead is too big. Furthermore, is file descriptor has a certain cost in the kernel and therefore the number is limited.

What is therefore needed is a kind of bus used to carry the notifications for many requests. A mechanism like `netlink` would be usable. The `netlink` sockets receive broadcast traffic for all the listeners and each process has to filter out the data which it is interested in. Broadcasting makes `netlink` sockets unattractive (at best) for event handling. The possible volume of notifications might be overwhelming. The overhead for the unnecessary wake-ups could be tremendous.

If filtering is accepted as not being a viable implementation requirement we have as a requirement for the solution that each process can create multiple, independent event channel, each capable of carrying arbitrarily many notification events from multiple sources. If we would not be able to create multiple independent channels a program could not concurrently and uncoordinatedly create such channels.

Each channel could be identified by a descriptor. This would then allow the use of the notification APIs in as many places as necessary independently. At each site only the relevant events are reported which allows the event handling to be as efficient as possible.

An event is not just an impulse, it has to transmit some information. The request which caused the event has to be identified. It is usually[2] regarded best to allow the programmer add additional information. A single pointer is sufficient, it allows the programmer to refer to additional data allocated somewhere else. There is no need to allow adding an arbitrary amount of data. The event data structure can therefore be of fixed length. This simplifies the event implementation and possibly allows it to perform better. If the transmission of the event structure would be implemented using socket the `SOCK_SEQPACKET` type can be used. The structure could look like this:

```
typedef struct event_data {
  enum { event_type_aio,
         event_type_msq,
         event_type_sig } ev_type;
  union {
    aio_ctx_t *ev_aio;
    mqd_t *ev_msq;
    sigevent_t ev_sig;
  } ev_un;
  ssize_t ev_result;
  int ev_errno;
  void *ev_data;
} event_data_t;
```

This structure can be used to signal events other than AIO completion. It could be a general mechanism. For instance, there currently is no mechanism to integrate POSIX message queues into `poll()` loops. With an extension to the `sigevent` structure it could be possible to register the event channel using the `mq_notify()` interface. The kernel can be extended to send events in all kinds of situations.

One possible implementation consists of introducing a new protocol family `PF_EVENT`. An event channel could then be created with:

```
int efd = socket(PF_EVENT,
     SOCK_SEQPACKET, 0);
```

[2]See the `sigevent` structure.

```
int ev_send(int s, const void *buf, size_t len, int flags, ev_t ec,
  void *data);
int ev_sendto(int s, const void *buf, size_t len, int flags,
  const struct sockaddr *to, socklen_t tolen, ev_t ec, void *data);
int ev_sendmsg(int s, const struct msghdr *msg, int flags, ev_t ec,
  void *data);
int ev_recv(int s, void *buf, size_t len, int flags, ev_t ec,
  void *data);
int ev_recvfrom(int s, void *buf, size_t len, int flags,
  struct sockaddr *to, socklen_t tolen, ev_t ec, void *data);
int ev_recvmsg(int s, struct msghdr *msg, int flags, ev_t ec,
  void *data);
```

Figure 1: Network Interfaces with Event Channel Parameters

The returned handle could be used in `poll()` calls and be used as the handle for the event channel. There are two potential problems which need some thought:

- The kernel cannot allow the event queue to take up arbitrary amounts of memory. There has to be an upper limit on the number of events which can be queued at the same time. When this happens a special event should be generated. It might be possible to use out-of-band notification for this so that the error is recognized right away.
- The number of events on a channel can potentially be high. In this case the overhead of all the `read()`/`recv()` calls could be a limiting factor. It might be beneficial to apply some of the techniques for the network I/O discussed in the next section to this problem as well. Then it might be possible to poll for new events without the system call overhead.

To enable optimizations like possible userlevel-visible event buffers the actual interface for the event handling should be something like this:

```
ec_t ec_create(unsigned flags);
int ec_destroy(ec_t ec);
int ec_to_fd(ec_t ec);
int ec_next_event(ec_t ec,
  event_data_t *d);
```

The `ec_to_fd()` function returns a file descriptor which can be used in `poll()` or `select()` calls. An implementation might choose to make this interface basically a no-op by implementing the event channel descriptor as a file descriptor. The `ec_next_event()` function returns the next event. A call might result in a normal `read()` or `recv` call but it might also use a user-level-visible buffer to avoid the system call overhead. The events signaled by `poll()` etc can be limited to the arrival of new data. I.e., the userlevel code is responsible for clearing the buffers before waiting for the next event using `poll()`. The kernel is involved in the delivery of new data and therefore this type of event can be quite easily be generated.

Handles of type `ec_t` can be passed to the asynchronous interfaces. The kernel can then

```
int aio_send(struct aiocb *aiocbp, int flags);
int aio_sendto(struct aiocb *aiocbp, int flags,
  const struct sockaddr *to, socklen_t tolen);
int aio_sendmsg(struct aiocb *aiocbp, int flags);
int aio_recv(struct aiocb *aiocbp, int flags);
int aio_recvfrom(struct aiocb *aiocbp, int flags, struct sockaddr *to,
  socklen_t tolen);
int aio_recvmsg(struct aiocb *aiocbp, int flags);
```

Figure 2: Network Interfaces matching POSIX AIO

create appropriate events on the channel. There will be no fixed relationship between the file descriptor or socket used in the asynchronous operation and the event channel. This gives the most flexibility to the programmer.

5 I/O Interfaces

There are several possible levels of innovation and complexity which can go into the design of the asynchronous I/O interfaces. It makes sense to go through them in sequence of increasing complexity. The more complicated interfaces will likely take advantage of the same functionality the less complicated need, too. Mentioning the new interfaces here is not meant to imply that all interfaces should be provided by the implementation.

The simplest of the interfaces can extend the network interfaces with asynchronous variants which use the event handling introduced in the previous section. One possibility is to extend interfaces like `recv()` and `send()` to take additional parameters to use event channels. The result is seen in Figure 1.

Calls to these functions immediately return. Valid requests are simply queued and the notifications about the completion are sent via the event channel `ec`. The `data` parameter is the additional value passed back as part of the `event_data_t` object read from the event channel. The event notification would signal the type of operation by setting `ev_type` appropriately. Success and the amount of data received or transmitted are stored in the `ev_errno` and `ev_result` elements.

There are two objections to this approach. First, the other frequently used interfaces for sockets (`read()` and `write()`) are not handled. Although their functionality is a strict subset of `recv()` and `send()` respectively it might be a deterrent. The second argument is more severe: there is no justification to limit the event handling to network transfer. The same functionality would be "nice to have"^{TM} for file, pipe, and FIFO I/O. Extending the `read()` and `write()` interfaces in the same way as the network I/O interfaces makes no sense, though. We already have interfaces which could be extended.

With a simple extension of the `sigevent` structure we can reuse the POSIX AIO interfaces. All that would be left to do is to define appropriate versions of the network I/O interfaces to match the existing POSIX AIO interfaces and change the `aiocb` structure slightly. The new interfaces can be seen in Figure 2. The `aiocb` structure needs to have one additional element:

```
struct aiocb {
  ...
  struct msghdr *aio_msg;
  ...
};
```

It is used in the `aio_sendmsg()` and `aio_recvmsg()` calls. The implementation can chose to reuse the memory used for the `aio_buf` element because it never gets used at the same time as `aio_msg`. The other four interfaces use `aio_buf` and `aio_nbytes` to specify the source and destination buffer respectively.

The `<signal.h>` header has to be extended to define `SIGEV_EC`. If the `sigev_notify` element of the `sigevent` structure is set to this value the completion is signal by an appropriate event available on an event channel. The channel is identified by a new element which must be added to the `sigevent` structure:

```
struct sigevent {
  ...
  ec_t sigev_ec;
  ...
};
```

The additional pointer value which is passed back to the application is also stored in the `sigevent` structure. The application has to store it in `sigev_value.sival_ptr` which is in line with all the other uses of this part of the `sigevent` structure.

Introducing these additional AIO interfaces and the `SIGEV_EC` notification mechanism would help to solve some problems.

- programs could get more efficient notification of events (at least more efficient than signals and thread creation), even for file I/O;
- network operations which require the extended functionality of the `recv` and `send` interfaces can be performed asynchronously;
- by pre-posting buffers with `aio_read()` or the `aio_recv()` and now the `aio_recv` and `aio_send` interfaces network I/O might be able to avoid intermediate buffers.

Especially the first two points are good arguments to implement these interfaces or at the very least allow the existing POSIX AIO in interfaces use the event channel notification. As explained in section section 2 the memory handling of the POSIX AIO functions makes direct use by the network hardware cumbersome and slower than necessary. Additionally the system call overhead is high when many interfaces use the event channel notification. As explained network requests have to be submitted. This can potentially be solved by extending the `lio_listio()` interface to allow submit multiple requests at once. But this will not solve the problem of the resulting event notification storm. For this we need more radical changes.

6 Advanced I/O Interfaces

For the more advanced interfaces we need to integrate the DMA memory handling into the I/O interfaces. We need to consider synchronous and asynchronous interfaces. We could ignore the synchronous interfaces and require the use of `lio_listio` or an equivalent interface but this is a bit cumbersome to use.

```
int dma_assoc(int sock, dma_mem_t mem, size_t size, unsigned flags);
int dma_disassoc(int sock, dma_mem_t, size_t size);
```

Figure 3: Association of DMA-able memory to Sockets

```
int sio_reserve(dma_mem_t dma, void **memp off, size_t size);
int sio_release(dma_mem_t dma, void *mem, size_t size);
```

Figure 4: Network Buffer Memory Management

For network interfaces it is ideally the interface which controls the memory into which incoming data is written. Today this happens with buffers allocated by and under full control of the kernel. It is conceivable to allow applications to allocate buffers and assign them to a given interface. This is where `dma_alloc()` comes in. The latter possibility has some distinct advantages; mainly, it gives the program the opportunity to influence the address space layout. This can be necessary for some programs.[3]

It is usually not possible to associate each network interface with a userlevel process. The network interface is in most cases a shared resource. The usual Unix network interface rules therefore need to be followed. A userlevel process opens a socket, binds the socket to a port, and it can send and receive data. For the incoming data the header decides which port the remote party wants to target. Based on the number, the socket is selected. Therefore the association of the DMA-able buffer should be with a socket. What is needed are interfaces as can be seen in Figure 3. It probably should be possible to associate more than one DMA-able memory region with a socket. This way it is possible to dynamically react to unexpected network traffic volume by adding additional buffers.

Once the memory is associated with the socket the application cannot use it anymore as it pleases until `dma_disassoc()` is called. The kernel has to be notified if the memory is written to and the kernel needs to tell the application when data is available to be read. Otherwise the kernel might start using a DMA memory region which the program is also using, thus overwriting the data. We therefore need at least interfaces as shown in Figure 4. The `sio_reserve()` interface allows to reserve (parts of) the DMA-able buffer for writing by the application. This will usually be done in preparation of a subsequent send operation. The `dma` parameter is the value returned by a previous call to `dma_alloc()`. We use a `size` parameter because this allows the DMA-able buffer to be split into several smaller pieces. As explained in section 3 it is more efficient to allocate larger blocks of DMA-able memory instead of many smaller ones because memory locking only works with page granularity. The implementation is responsible for not using the same part of the buffer more than once at the same time. A pointer to the available memory is returned in the variable pointed to by `memp`.

When reading from the network the situation is reversed: the kernel will allocate the mem-

[3]For instance, when address space is scarce or when fixed addresses are needed.

```
int sio_send(int sock, const void *buf, size_t size, int flags);
int sio_sendto(int sock, const void *buf, size_t size, int flags,
  const struct sockaddr *to, socklen_t tolen);
int sio_sendmsg(int sock, const void *buf, size_t size, int flags);
int sio_recv(int sock, void **buf, size_t size, int flags);
int sio_recvfrom(int sock, const void **buf, size_t size, int flags,
  struct sockaddr *to, socklen_t tolen);
int sio_recvmsg(int sock, const void **buf, size_t size, int flags);
```

Figure 5: Advanced Synchronous Network Interfaces

ory region into which it stores the incoming data. This happens using the kernel-equivalent of the `sio_reserve()` interface. Then the program is notified about the location and size of the incoming data. Until the program is done handling the data the buffer cannot be reused. To signal that the data has been handled, the `sio_release()` interface is used. It is also possible to use the interface to abort the preparation of a write operation by undoing the effects of a previous `sio_reserve()` call.

The `sio_reserve()` and `sio_release()` interfaces basically implement dynamic memory allocation and deallocation. It adds an undue burden on the implementation to require a full-fledged `malloc`-like implementation. It is therefore suggested to require a significant minimum allocation size. If reservations are also rounded according to the minimum size this will in turn limit the number of reservations which can be given out at any given time. It is possible to use a simple bitmap allocator.

What remains to be designed are the actual network interfaces. For the synchronous interfaces we need the equivalent of the `send` and `recv` interfaces. The `send` interfaces can basically work like the existing Unix interfaces with the one exception that the memory block containing the data must be part of a DMA-able memory region. The `recv` interfaces need to have one crucial difference: the implementation must be able to decide the location of the buffer containing the returned data. The resulting interfaces can be seen in Figure 5.

The programmer has to make sure the buffer pointers passed to the `sio_send` functions have been returned by a `sio_reserve()` call or as part of the notification of a previous `sio_recv` call. The implementation can potentially detect invalid pointers.

When the `sio_recv` functions return, the pointer pointed to by the second parameter contains the address of the returned data. This address is in the DMA-able memory area associated with the socket. After the data is handled and the buffer is not used anymore the application has to mark the region as unused by calling `sio_release()`. Otherwise the kernel would run out of memory to store the incoming data in.

For the asynchronous interfaces one could imagine simply adding a `sigevent` structure parameter to the `sio_recv` and `sio_send` interfaces. This is unfortunately not sufficient. The program must be able to retrieve the error status and the actual number of bytes which have been received or sent. There is no way to transmit this information in the `sigevent` structure. We could extend it but would duplicate functionality which is already available. The asynchronous file I/O interfaces have the same problem and the solution is the AIO control block structure aiocb. It only makes sense to extend the POSIX AIO interfaces. We already defined the additional interfaces needed in Figure 2. What is missing is the tie-in with the DMA handling.

For this the most simplistic approach is to extend aiocb structure by adding an element aio_dma_buf of type dma_mem_t replacing the aio_buf pointer for DMA-ready operations. To use aio_dma_buf instead of aio_buf the caller passes the new AIO_DMA_BUF flag to the aio_recv and aio_send interfaces. For the lio_listio() interface it is possible to define new operations LIO_DMA_READ and LIO_DMA_WRITE. This leaves the existing aio_read() and aio_write() interfaces. It would be possible to define alternative interfaces which take a flag parameter or one could simply ignore the problem and tell people to use lio_listio() instead.

The implementation of the AIO functions to receive data when operating on DMA-able buffers could do more than just pass the request to the kernel. The implementation can keep track of the buffers involved and check for available data in them before calling the kernel. If data is available the call can be avoided and the appropriate buffer can be made known through an appropriate event. When writing the data could be written into the DMA-able buffer (if necessary). Depending on the implementation of the user-level/kernel interaction of the DMA-able buffers it might or might not be necessary to make a system call to notify the kernel about the new pending data.

7 Related Interfaces

The event channel mechanism is general enough to be used in other situations than just I/O. They can help solving a long-standing problem of the interfaces Unix systems provide. Programs, be it server or interactive programs, are often designed with a central loop from which the various activities requested are initiated. There can be one thread working the inner loop or many. The requested actions can be performed by the thread which received the request or a new thread can be created which performs the action. The threads in the program are then either waiting in the main loop or busy working on an action. If the action could potentially be delayed significantly the thread would add the wait event to the list the main loop handles and then enters the main loop again. This achieves maximum resource usage.

In reality this is not so easy. Not all events can be waited on with the same mechanism. POSIX does not provide mechanisms to use poll() to wait for messages to arrive in message queues, for mutexes to be unlocked, etc. This is where the event channels can help. If we can associate an event channel with these objects the kernel could generate events whenever the state changes.

For POSIX message queues there is fortunately not much which needs to be done. The mq_notify() interface takes a sigevent structure parameter. Once the implementation is extended to handle SIGEV_EC for I/O it should work here, too. One question to be answered is what to pass as the data parameter which can be used to identify the request.

For POSIX semaphore we need a new interface to initiate asynchronous waiting. Figure 6 shows the prototype for sem_await(). The first two parameters are the same as for sem_wait(). The latter two parameters specify the event channel and the parameter to pass back. When the event reports a successful operation the semaphore has been posted. It is not necessary to call sem_wait() again.

```
int sem_await(sem_t semdes, const struct timespec *abstime,
              ec_t ec, void *data);
int pthread_mutex_alock(pthread_mutex_t *mutex, ec_t ec, void *data);
```

Figure 6: Additional Event Channel Users

The actual implementation of this interface will be more interesting. Semaphores and also mutexes are implemented using futexes. Only part of the actual implementation is in the kernel. The kernel does not know the actual protocol used for the synchronization primitive, this is left to the implementation. In case the event channel notification is requested the kernel will have to learn about the protocol.

Once the POSIX semaphore problem is solved it is easy enough to add support for the POSIX mutexes, read-write mutexes, barriers, etc. The `pthread_mutex_alock()` interface is Figure 6 is a possible solution. The other synchronization primitives can be similarly handled. This extends also to the System V message queues and semaphores. The difference for the latter two is that the implementation is already completely in the kernel and therefore the implementation should be significantly simpler.

8 Summary

The proposed interfaces for network I/O have the potential of great performance improvements. They avoid using the most limiting resources in a modern computer: memory-to-CPU cache and CPU cache-to-memory bandwidth. By minimizing the number of copies which have to be performed the CPUs have the chance of keeping up with the faster increasing network speeds.

Along the way we sketched out a event handling implementation which is not only efficient enough to keep up with the demands of the network interfaces. It is also versatile enough to finally allow implementing a unified inner loop of all event driven programs. With `poll` or `select` interfaces being able to receive event notifications for currently unobservable objects like POSIX/SysV message queues and futexes many programs have the opportunity to become much easier because special handling for these cases can be removed.

Problem Solving With Systemtap

Frank Ch. Eigler
Red Hat
`fche@redhat.com`

Abstract

Systemtap is becoming a useful tool to help solve low-level OS problems. Most features described in the future tense at last year's OLS are now complete. We review the status and recent developments of the system. In passing, we present solutions to some complex low-level problems that bedevil kernel and application developers.

Systemtap recently gained support for static probing markers that are compiled into the kernel, to complement the dynamic `kprobes` system. It is a simple and fast mechanism, and we invite kernel developers and other trace-like tools to adopt it.

1 Project status

At OLS 2005, we presented[4] systemtap, the open source tool being developed for tracing/probing of a live unmodified linux system. It accepts commands in a simple scripting language, and hooks them up to probes inserted at requested code locations within the kernel. When the kernel trips across the probes, routines in a compiled form of the script are quickly run, then the kernel resumes. Over the last year, with the combined efforts of a dozen developers supported by four companies, much of this theory has turned into practice.

1.1 Scripting language

The systemtap script is a small domain-specific language resembling `awk` and C. It has only a few data types (integers and strings, plus associative arrays of these), full control structures (blocks, conditionals, loops, functions). It is light on punctuation (semicolons optional) and on declarations (types are inferred and checked automatically). Its core concept, the "probe," consists of a probe point (its trigger event) and its handler (the associated statements).

Probe points name the kernel events at which the statements should be executed. One may name nearly any function, or a source file and line number where the breakpoint is to be set (just like in a symbolic debugger), or request an asynchronous event like a periodic timer. Systemtap defines a hierarchical probe point namespace, a little like DNS.

Probe handlers have few constraints. They can print data right away, to provide a sort of on-the-fly `printk`. Or, they can save a timestamp in a variable and compare it with a later probe hit, to derive timing profiles. Or, they can follow kernel data structures, and speak up if something is amiss.

The scripting language is implemented by a translator that creates C code, which is in turn compiled into a binary kernel module. Probe points are mapped to virtual addresses by reference to the kernel's DWARF debugging information left over from its build. The same data is used to resolve references to kernel "target-side" variables. Their compiled nature allows even elaborate probe scripts to run fast.

Safety is an essential element of the design. All the language constructs are subjected to translation- and run-time checks, which aim to prevent accidental damage to the system. This includes prevention of infinite loops, memory use, and recursion, pointer faults, and several others. Many checks may be inspected within the translator-generated C code.

Some safety mechanisms are incomplete at present. Systemtap contains a blacklist of kernel areas that are deemed unsafe to probe, since they might trigger infinite probing recursion, locking reentrancy, or other nasty phenomena. This blacklist is just getting started, so probing using broad wildcards is a recipe for panics. Similarly, we haven't sufficiently analyzed the script-callable utility functions like our `gettimeofday` wrapper to ensure that it is safe to call from any probe handler. Work in these directions is ongoing.

1.2 Recent developments

The most basic development since last summer is that the system *works*, whereas last year we relied on several mock-ups. You can download it[1], build it, and use it on your already installed kernels today. It is not perfect nor complete, but nor is it vapourware.

kprobes has received a heart transplant. The most significant of these was truly concurrent probing on multiprocessor machines, made possible by a switch to RCU data structures. Implementation details are discussed in a separate paper [3] during this conference. In order to exploit the parallelism enabled by this improvement, systemtap supports variables to track global *statistics aggregates* like averages or counts using contention-free data structures.

For folks who like the exhilaration of full control, or have a distaste for the scripting language, Systemtap supports bypassing the cushion. In "guru mode," systemtap allows intermingling of literal C code with script, to go beyond the limitations of pure script code. One can query or manipulate otherwise inaccessible kernel state directly, but bears responsibility for doing so *safely*.

Systemtap documentation is slowly growing, as is our collection of sample scripts. There is a fifteen-page language tutorial, and a few dozen worked out examples on our web site. More and more first-time users are popping up on the mailing list, so we are adapting to supporting new users, not just fellow project developers.

1.3 Usage scenarios

While it's still early, systemtap has suggested several uses. First is simple exploration and profiling. A probe on "timer.profile" and collecting stack backtrace samples gets one a coarse profile. A probe at a troubled function, with a similar a stack backtrace, tells one who is the troublemaker. Probes on system call handling functions (or more conveniently named aliases defined in a library) give one an instant system-wide `strace`, with as much filtering and summarizing as one may wish. As a taste, figure 1 demonstrates probing function nesting within a compilation unit.

Daniel Berranger [1] arranged to run systemtap throughout a Linux boot sequence (`/etc/init.d` scripts) to profile the I/O

[1]`http://sourceware.org/systemtap/`

and forking characteristics of the many startup scripts and daemons. Some wasteful behavior showed up right away in the reports. On a similar topic, Dave Jones [2] is presenting a paper at this conference.

Another problem may be familiar: an overactive `kswapd`. In an old Red Hat Enterprise Linux kernel, it was found that some inner page-scanning loop ran several orders of magnitude more iterations than anticipated, due to some error in queue management code. Does this kind of thing not happen regularly? Systemtap was not available for diagnosing this bug, but it would have been easy to probe loops in the suspect functions, say by source file and line number, to count and graph relative execution counts.

2 Static probing markers

Systemtap recently added support for *static probing markers* or "markers" for short. This is a way of letting developers designate points in their functions as being candidates for systemtap-style probing. The developer inserts a macro call at the points of interest, giving the marker a name and some optional parameters, and grudgingly recompiles the kernel. (The name can be any alphanumeric symbol, and should be reasonably unique across the kernel or module. Parameters may be string or numeric expressions.)

In exchange for this effort, systemtap marker-based probes are faster and more precise than kprobes. The better precision comes from not having to covet the compiler's favours. Such fickle favours include retaining clean boundaries in the instruction stream between interesting statements, and precisely describing positions of variables in the stack frame. Since markers don't rely on debugging information, neither favour is required, and the compiler can channel its charms into unabated optimization. The speed advantage comes from using direct call instructions rather than `int 3` breakpoints to dispatch to the systemtap handlers. We will see below just how big a difference this makes.

```
STAP_MARK (name);
STAP_MARK_NS (name,num,string);
```

Just putting a marker into the code does nothing except waste a few cycles. A marker can be "activated" by writing a systemtap probe associated with the marker name. All markers with the same name are identified, and are made to call the probe handler routine. Like any other systemtap probe, the handler can trace, collect, filter, and aggregate data before returning.

```
probe kernel.mark("name") { }
probe module("drv").mark("name") { }
```

2.1 Implementation

As hinted above, the probe marker is a macro[2] that consists of a conditional indirect function call. Argument expressions are evaluated in the conditional function call. Similarly to C++, an explicit argument-type signature is appended to the macro and the static variable name.

```
#define STAP_MARK(n) do { \
  static void (*__mark_##n##_)(); \
  if (unlikely (__mark_##n##_)) \
    (void) (__mark_##n##_()); \
} while (0)
```

In x86 assembly language, this translates to a load from a direct address, test, and a conditional branch over a call sequence. The

[2]Systemtap includes a header file that defines a scores of type/arity permutations.

load/zero-test is easily optimized by "hoisting" it up (earlier), since it is operating on private data. With GCC's `-freorder-blocks` optimization flag, the instructions for the function call sequence tend to be pushed well away from (beyond) the hot path, and get jumped to using a conditional forward branch. That is ideal from the perspective of hardware static branch prediction.

A new static variable is created for each macro. If the macro is instantiated within an inline function, all inlined instances within a program will share that same variable. Systemtap can search for the variables in the symbol table by matching names against the stylized naming scheme. Further, systemtap deduces argument types from the signature suffix, so it can write a type-safe function to accept the parameters and dispatch to a compiled probe handler.

During probe initialization, the static variable containing the marker's function pointer is simply overwritten to point at the handler, and it is cleared again at shutdown.[3]

This design implies that only a single handler can be associated with any single marker: other systemtap sessions are locked out temporarily. Should this become a problem for particularly popular markers, we can add support for "multi-marker" macros that use some small number of synonymous static variables instead of one. This would trade utility for speed.

2.2 Performance

Several performance metrics are interesting: code bloat, slowdown due to a dormant marker, dispatch cost of an active marker. These quantities may be compared to the classic kprobes alternative. On all these metrics, markers seem to perform well.

[3]These operations atomically synchronize using `cmpxchg`.

For demonstration purposes, we inserted marker macros in just two spots in a 2.6.16-based kernel: the scheduler context-switch routine, just before `switch_to` (passing the "from" and "to" `task->pid` numbers), and the system call handler `sys_getuid` (passing `current->uid`). All tests were run on a Fedora Core 5 machine with a 3 GHz Pentium 4 HT.

Code bloat is the number of bytes of instruction code needed to support the marker, which impacts the instruction cache. With kprobes, there is no code inserted, so those numbers are zero. We measured it for static markers by disassembling otherwise identical kernel binaries, compiled with and without markers.

function	test	call
getuid	10	19
context_switch	19	34

Slowdown due to a dormant marker is the time penalty for having a potential but unused probe point. This quantity is also zero for kprobes. For our static markers, it is the time taken to test whether the static variable is set, and it being clear, to bypass the probe function call. It may incur a data cache miss (for loading the static variable), but the actual test and properly predicted branch can be nearly "free."

Indeed, a microbenchmark that calls an marker-instrumented `getuid` system call in a tight loop a million times has minimum and average times that match one that calls an uninstrumented system call (`getgid`). A different microbenchmark that runs the same marker macro but in user space, surrounded by `rdtscll` calls, indicates a cost of a handful of cycles each: 4–20.

Since the slowdown due to a dormant marker is so small, we plan to measure a heavily instrumented kernel macroscopically. However, adding markers strategically into the kernel is challenging, if they are to represent plausible extra load.

Finally, let's discuss the dispatch speed of an active marker. This is important because it relates inversely to the maximum number of probes that can trigger per unit time. The overhead for a reasonable frequency of probe hits should not overwhelm the system. For our static markers, the dispatch overhead consists of the indirect function call. On the test platform, this additional cost is just 50–60 cycles.

For kprobes, an active probe includes an elaborate process involving triggering a breakpoint fault (`int 3` on x86), entering the fault handler, identifying which handler belongs to that particular breakpoint address, calling the handler, single-stepping the original instruction under the breakpoint, and probably some other steps we left out.

A realistic systemtap-based microbenchmark measured the time required for one round trip of the same functions used above: the marker-instrumented `sys_getuid` and uninstrumented `sys_getgid`. Each probe handler is identical, and increments a script-level global counter variable for each visit. The following matrix summarizes the typical number of nanoseconds per system call (lower is better) with the listed instrumentation active.

function	marker	kprobe	both	neither
getuid	820	2100	2250	620
getgid		2100		620

Note that the complete marker-based probes run in 200 ns, and kprobes-based probes run in 1480 ns. Some arithmetic lets us work backward, to estimate just the dispatching times and exclude the systemtap probes. The cost of the 50–60 cycles of function call dispatch for the markers (measured earlier) takes about 20 ns on the test host. That implies that the systemtap probe handler took about 180 ns. Since identical probe handlers were run for both kprobes and markers, we can subtract that, leaving 1300 ns as the kprobes dispatch overhead.

While the above analysis only pretends to be quantitative, it gives some evidence that markers have attractive performance: cheap to sit around dormant, and fast when activated.

3 Next steps

3.1 User-space probes

Systemtap still lacks support for probing user-space programs: we can go no higher than the system call interface. A kprobes extension is under development to allow the same sorts of breakpoints to be inserted into shared libraries and executables at runtime that it now manages in the kernel. When this part is finished and accepted, systemtap will exploit it shortly. Probes in user space would use a similar syntax to refer to sources or symbols as already available for kernel probe points.

Probing in user space may seem like a task for a different sort of tool, perhaps a plain debugger like gdb, or a fancier one like frysk[4], or another supervisor process based on `ptrace`. However, we believe that the handler routine of even a user-space probe should run in kernel space, because:

1. The microsecond level speed of a kprobes "round trip" is still an order of magnitude faster than the equivalent process state query / manipulation using the ptrace API.

[4]`http://sources.redhat.com/frysk`

2. Some problems require correlation of activities in the kernel with those in user-space. Such correlations are naturally expressed by a single script that shares variables amongst kernel- and user-space probes.

Once we pass that hurdle, joint application of user-space kprobes and static probing markers will make it possible for user-space programs and libraries to contain probing markers too. This would let libraries or programs designate their own salient probe points, while enjoying a low dormant probe cost. Language interpreters like Perl and PHP can insert markers into their evaluation loops to mark events like script function entries/exits and garbage collection. Complex applications can instrument multithreading events like synchronization and lock contention.

3.2 Debugging aid

Systemtap is becoming stable enough that kernel developers should feel comfortable with using it as a first-ditch debugging aid. When you run into a problem where a little bit of tracing, profiling, event counting might help, we are eager to help you write the necessary scripts.

3.3 Sysadmin aid

We would like to develop a suite of systemtap scripts that supplant tools like `netstat`, `vmstat`, `strace`. For inspiration, it may be desirable to port the OpenSolaris DTrace-Toolkit[5], which is a suite of dtrace scripts to provide an overview of the entire system's activity. Systemtap will make it possible to save and reuse *compiled* scripts, so that deployment and execution of such a suite could be easier and faster.

[5]`http://www.opensolaris.org/os/community/dtrace/dtracetoolkit/`

3.4 Grand unified tracing

There are many linux kernel tracing projects around. Every few months, someone reinvents LTT and auditing. While the author does not understand all the reasons for which these tools tend not to be integrated into the mainstream kernel, perhaps one of them is performance.

To the extent that is true, we propose that these groups consider using a shared pool of static markers as the basic kernel-side instrumentation mechanism. If they prove to have as low dormant cost and as high active performance as initial experience suggests, perhaps this could motivate the various tracing efforts and kernel subsystem developers to finally join forces. Let's designate standard trace/probe points once and for all. Tracing backends can attach to these markers the same way systemtap would. There would be no need for them to maintain kernel patches any more. Let's think about it.

References

[1] Daniel Berranger. `http://people.redhat.com/berrange/systemtap/bootprobe/`, January 2006.

[2] Dave Jones. Why Userspace Sucks. In *Proceedings of the 2006 Ottawa Linux Symposium*, July 2006.

[3] Ananth N. Mavinakayanahalli et al. Probing the Guts of Kprobes. In *Proceedings of the 2006 Ottawa Linux Symposium*, July 2006.

[4] Vara Prasad et al. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2005 Ottawa Linux Symposium*, volume 2, pages 49–64, July 2005.

```
# cat socket-trace.stp
probe kernel.function("*@net/socket.c") {
  printf ("%s -> %s\n", thread_indent(1), probefunc())
}
probe kernel.function("*@net/socket.c").return {
  printf ("%s <- %s\n", thread_indent(-1), probefunc())
}

# stap socket-trace.stp
     0 hald(2632): -> sock_poll
    28 hald(2632): <- sock_poll
[...]
     0 ftp(7223): -> sys_socketcall
  1159 ftp(7223):  -> sys_socket
  2173 ftp(7223):   -> __sock_create
  2286 ftp(7223):    -> sock_alloc_inode
  2737 ftp(7223):    <- sock_alloc_inode
  3349 ftp(7223):    -> sock_alloc
  3389 ftp(7223):    <- sock_alloc
  3417 ftp(7223):   <- __sock_create
  4117 ftp(7223):   -> sock_create
  4160 ftp(7223):   <- sock_create
  4301 ftp(7223):   -> sock_map_fd
  4644 ftp(7223):    -> sock_map_file
  4699 ftp(7223):    <- sock_map_file
  4715 ftp(7223):   <- sock_map_fd
  4732 ftp(7223):  <- sys_socket
  4775 ftp(7223): <- sys_socketcall
[...]
```

Figure 1: Tracing and timing functions in `net/sockets.c`.

Perfmon2: a flexible performance monitoring interface for Linux

Stéphane Eranian
HP Labs
`eranian@hpl.hp.com`

Abstract

Monitoring program execution is becoming more than ever key to achieving world-class performance. A generic, flexible, and yet powerful monitoring interface to access the performance counters of modern processors has been designed. This interface allows performance tools to collect simple counts or profiles on a per kernel thread or system-wide basis. It introduces several innovations such as customizable sampling buffer formats, time or overflow-based multiplexing of event sets. The current implementation for the 2.6 kernel supports all the major processor architectures. Several open-source and commercial tools based on interface are available. We are currently working on getting the interface accepted into the mainline kernel. This paper presents an overview of the interface.

1 Introduction

Performance monitoring is the action of collecting information about the execution of a program. The type of information collected depends on the level at which it is collected. We distinguish two levels:

- the program level: the program is instrumented by adding explicit calls to routines that collect certain metrics. Instrumentation can be inserted by the programmer or the compiler, e.g., the `-pg` option of GNU cc. Tools such as HP Caliper [5] or Intel PIN [17] can also instrument at runtime. With those tools, it is possible to collect, for instance, the number of times a function is called, the number of time a basic block is entered, a call graph, or a memory access trace.

- the hardware level: the program is not modified. The information is collected by the CPU hardware and stored in performance counters. They can be exploited by tools such as OProfile and VTUNE on Linux. The counters measure the micro-architectural behavior of the program, i.e., the number of elapsed cycles, how many data cache stalls, how many TLB misses.

When analyzing the performance of a program, a user must answer two simple questions: where is time spent and why is spent time there? Program-level monitoring can, in many situations and with some high overhead, answer the first, but the second question is best solved with hardware-level monitoring. For instance, `gprof` can tell you that a program spends 20% of its time in one function. The difficulty is to know why. Is this because the function is called a lot? Is this due to algorithmic problems? Is it because the processor stalls? If so, what is causing the stalls? As this simple example shows, the two levels of monitoring can be complementary.

The current CPU hardware trends are increasing the need for powerful hardware monitoring. New hardware features present the opportunity to gain considerable performance improvements through software changes. To benefit from a multi-threaded CPU, for instance, a program must become multi-threaded itself. To run well on a NUMA machine, a program must be aware of the topology of the machine to adjust memory allocations and thread affinity to minimize the number of remote memory accesses. On the Itanium [3] processor architecture, the quality of the code produced by compilers is a big factor in the overall performance of a program, i.e, the compiler must extract the parallelism of the program to take advantage of the hardware.

Hardware-based performance monitoring can help pinpoint problems in how software uses those new hardware features. An operating system scheduler can benefit from cache profiles to optimize placement of threads to avoiding cache thrashing in multi-threaded CPUs. Static compilers can use performance profiles to improve code quality, a technique called Profile-Guided Optimization (PGO). Dynamic compilers, in Managed Runtime Environments (MRE) can also apply the same technique. Profile-Guided Optimizations can also be applied directly to a binary by tools such as iSpike [11]. In virtualized environments, such as Xen [14], system managers can also use monitoring information to guide load balancing. Developers can also use this information to optimize the layout of data structures, improve data prefetching, analyze code paths [13]. Performance profiles can also be used to drive future hardware requirements such as cache sizes, cache latencies, or bus bandwidth.

Hardware performance counters are logically implemented by the Performance Monitoring Unit (PMU) of the CPU. By nature, this is a fairly complex piece of hardware distributed all across the chip to collect information about key components such as the pipeline, the caches, the CPU buses. The PMU is, by nature, very specific to each processor implementation, e.g., the Pentium M and Pentium 4 PMUs [9] have not much in common. The Itanium processor architecture specifies the framework within which the PMU must be implemented which helps develop portable software.

One of the difficulties to standardize on a performance monitoring interface is to ensure that it supports all existing and future PMU models without preventing access to some of their model specific features. Indeed, some models, such as the Itanium 2 PMU [8], go beyond just counting events, they can also capture branch traces, where cache misses occur, or filter on opcodes.

In Linux and across all architectures, the wealth of information provided by the PMU is oftentimes under-exploited because a lack of a flexible and standardized interface on which tools can be developed.

In this paper, we give an overview of *perfmon2*, an interface designed to solve this problem for all major architectures. We begin by reviewing what Linux offers today. Then, we describe the various key features of this new interface. We conclude with the current status and a short description of the existing tools.

2 Existing interfaces

The problem with performance monitoring in Linux is not the lack of interface, but rather the multitude of interfaces. There are at least three interfaces:

- OProfile [16]: it is designed for DCPI-style [15] system-wide profiling. It is supported on all major architectures and is enabled by major Linux distributions. It can generate a flat profile and a call graph per program. It comes with its own tool set, such as `opcontrol`. Prospect [18] is another tool using this interface.
- perfctr [12]: it supports per-kernel-thread and system-wide monitoring for most major processor architectures, except for Itanium. It is distributed as a stand-alone kernel patch. The interface is mostly used by tools built on top of the PAPI [19] performance toolkit.
- VTUNE [10]: the Intel VTUNE performance analyzer comes with its own kernel interface, implemented by an open-source driver. The interface supports system-wide monitoring only and is very specific to the needs of the tool.

All these interfaces have been designed with a specific measurement or tool in mind. As such, their design is somewhat limited in scope, i.e., they typically do one thing very well. For instance, it is not possible to use OProfile to count the number of retired instructions in a thread. The perfctr interface is the closest match to what we would like to build, yet it has some shortcomings. It is very well designed and tuned for self-monitoring programs but sampling support is limited, especially for non self-monitoring configurations.

With the current situation, it is not necessarily easy for developers to figure out how to write or port their tools. There is a question of functionalities of each interfaces and then, a question of distributions, i.e., which interface ships with which distribution. We believe this situation does not make it attractive for developers to build modern tools on Linux. In fact, Linux is lagging in this area compared to commercial operating systems.

3 Design choices

First of all, it is important to understand why a kernel interface is needed. A PMU is accessible through a set of registers. Typically those registers are only accessible, at least for writing, at the highest privilege level of execution (pl0 or ring0) which is where only the kernel executes. Furthermore, a PMU can trigger interrupts which need kernel support before they can be converted into a notification to a user-level application such as a signal, for instance. For those reasons, the kernel needs to provide an interface to access the PMU.

The goal of our work is to solve the hardware-based monitoring interface problem by designing a single, generic, and flexible interface that supports all major processor architectures. The new interface is built from scratch and introduces several innovations. At the same time, we recognize the value of certain features of the other interfaces and we try to integrate them wherever possible.

The interface is designed to be built into the kernel. This is the key for developers, as it ensures that the interface will be available and supported in all distributions.

To the extent possible, the interface must allow existing monitoring tools to be ported without many difficulties. This is useful to ensure undisrupted availability of popular tools such as VTUNE or OProfile, for instance.

The interface is designed from the bottom up, first looking at what the various processors provide and building up an operating system interface to access the performance counters in a uniform fashion. Thus, the interface is not designed for a specific measurement or tool.

There is efficient support for *per-thread* monitoring where performance information is collected on a kernel thread basis, the PMU state is saved and restored on context switch. There is also support for system-wide monitoring where all threads running on a CPU are monitored and the PMU state persists across context switches.

In either mode, it is possible to collect simple counts or profiles. Neither applications nor the Linux kernel need special compilation to enable monitoring. In per-thread mode, it is possible to monitor unmodified programs or multi-threaded programs. A monitoring session can be dynamically attached and detached from a running thread. Self-monitoring is supported for both counting and profiling.

The interface is available to regular users and not just system administrators. This is especially important for per-thread measurements. As a consequence, it is not possible to assume that tools are necessarily well-behaved and the interface must prevent malicious usage.

The interface provides a uniform set of features across platforms to maximize code re-use in performance tools. Measurement limitations are mandated by the PMU hardware not the software interface. For instance, if a PMU does not capture where cache misses occur, there is nothing the interface nor its implementation can do about it.

The interface must be extensible because we want to support a variety of tools on very different hardware platforms.

4 Core Interface

The interface leverages a common property of all PMU models which is that the hardware interface always consists of a set of configuration registers, that we call PMC (Performance Monitor Configuration), and a set of data registers, that we call PMD (Performance Monitor Data). Thus, the interface provides basic read/write access to the PMC/PMD registers.

Across all architectures, the interface exposes a uniform register-naming scheme using the PMC and PMD terminology inherited from the Itanium processor architecture. As such, applications actually operate on a logical PMU. The mapping from the logical to the actual PMU is described in Section 4.3.

The whole PMU machine state is represented by a software abstraction called a *perfmon context*. Each context is identified and manipulated using a file descriptor.

4.1 System calls

The interface is implemented with multiple system calls rather than a device driver. Per-thread monitoring requires that the PMU machine state be saved and restored on context switch. Access to such routine is usually prohibited for drivers. A system call provides more flexibility than `ioctl` for the number, type, and type checking of arguments. Furthermore, system calls reinforce our goal of having the interface be an integral part of the kernel, and not just an optional device driver.

The list of system calls is shown in Table 1. A context is created by the `pfm_create_context` call. There are two types of contexts: per-thread or system-wide. The type is determined when the context is created. The same set of functionalities is available to both types

```
int pfm_create_context(pfarg_ctx_t *c, void *s, size_t s)
int pfm_write_pmcs(int f, pfarg_pmc_t *p, int c)
int pfm_write_pmds(int f, pfarg_pmd_t *p, int c)
int pfm_read_pmds(int f, pfarg_pmd_t *p, int c)
int pfm_load_context(int f, pfarg_load_t *l)
int pfm_start(int fd, pfarg_start_t *s)
int pfm_stop(int f)
int pfm_restart(int f)
int pfm_create_evtsets(int f, pfarg_setdesc_t *s, int c)
int pfm_getinfo_evtsets(int f, pfarg_setinfo_t *i, int c)
int pfm_delete_evtsets(int f, pfarg_setdesc_t *s, int c)
int pfm_unload_context(int f)
```

Table 1: perfmon2 system calls

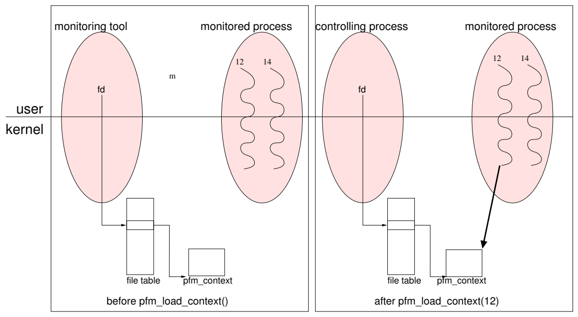


Figure 1: attaching to a thread

of context. Upon return from the call, the context is identified by a file descriptor which can then be used with the other system calls.

The write operations on the PMU registers are provided by the `pfm_write_pmcs` and `pfm_write_pmds` calls. It is possible to access more than one register per call by passing a variable-size array of structures. Each structure consists, at a minimum, of a register index and value plus some additional flags and bitmasks.

An array of structures is a good compromise between having a call per register, i.e., one register per structure per call, and passing the entire PMU state each time, i.e., one large structure per call for all registers. The cost of a system call is amortized, if necessary, by the fact that multiple registers are accessed, yet flexibility is not affected because the size of the array is variable. Furthermore, the register structure definition is generic and is used across all architectures.

The PMU can be entirely programmed before the context is attached to a thread or CPU. Tools can prepare a pool of contexts and later attach them on-the-fly to threads or CPUs.

To actually load the PMU state onto the actual hardware, the context must be bound to either a kernel thread or a CPU with the `pfm_load_context` call. Figure 1 shows the effect of the call when attaching to a thread of a dual-threaded process. A context can only be bound to one thread or CPU at a time. It is not possible to bind more than one context to a thread or CPU. Per-thread monitoring and system-wide monitoring are currently mutually exclusive. By construction, multiple concurrent per-thread contexts can co-exist. Potential conflicts are detected when the context is attached and not when it is created.

An attached context persists across a call to `exec`. On `fork` or `pthread_create`, the context is not automatically cloned in the new thread because it does not always make sense to aggregate results or profiles from *child* processes or threads. Monitoring tools can leverage the 2.6 kernel `ptrace` interface to receive notifications on the `clone` system call to decide whether or not to monitor a new thread or process. Because the context creation and attachment are two separate operations, it is possible to *batch* creations and simply attach and start on notification.

Once the context is attached, monitoring can be started and stopped using the `pfm_start` and `pfm_stop` calls. The values of the PMD registers can be extracted with the `pfm_read_pmds` call. A context can be detached with `pfm_unload_context`. Once detached the context can later be re-attached to any thread or CPU if necessary.

A context is destroyed using a simple `close` call. The other system calls listed in Table 1 relate to sampling or event sets and are discussed in later sections.

Many 64-bit processor architectures provide the ability to run with a *narrow* 32-bit instruction set. For instance, on Linux for x86_64, it is possible to run unmodified 32-bit i386 binaries. Even though, the PMU is very implementation specific, it may be interesting to develop/port tools in 32-bit mode. To avoid data conversions in the kernel, the perfmon2 ABI is designed to be portable between 32-bit (ILP32) and 64-bit (LP64) modes. In other words, all the data structures shared with the kernel use fixed-size data types.

4.2 System-wide monitoring

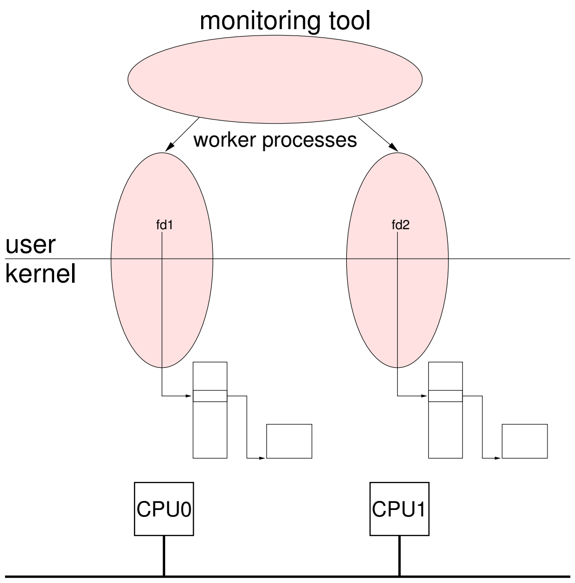


Figure 2: monitoring two CPUs

A perfmon context can be bound to only one CPU at a time. The CPU on which the call to `pfm_load_context` is executed determines the monitored CPU. It is necessary to set the affinity of the calling thread to ensure that it runs on the CPU to monitor. The affinity can later be modified, but all operations requiring access to the actual PMU must be executed on the monitored CPU, otherwise they will fail. In this setup, coverage of a multi-processor system (SMP), requires that multiple contexts be created and bound to each CPU to monitor. Figure 2 shows a possible setup for a monitoring tool on a 2-way system. Multiple non-overlapping system-wide attached context can co-exist.

The alternative design is to have the kernel propagate the PMU access to all CPUs of interest using Inter-Processor-Interrupt (IPI). Such approach does make sense if all CPUs are always monitored. This is the approach chosen by OProfile, for instance.

With the perfmon2 approach, it is possible to measure subsets of CPUs. This is very interesting for large NUMA-style or multi-core machines where all CPUs do not necessarily run the same workload. And even then, with a uniform workload, it possible to divide the CPUs into groups and capture different events in each group, thereby overlapping distinct measurements in one run. Aggregation of results can be done by monitoring tools, if necessary.

It is relatively straightforward to construct a user-level helper library that can simplify monitoring multiple CPUs from a single thread of control. Internally, the library can pin threads on the CPUs of interest. Synchronization between threads can easily be achieved using a barrier built with the POSIX-threads primitives. We have developed and released such a library as part of the `libpfm` [6] package.

Because PMU access requires the controlling thread to run on the monitored CPU, processor and memory affinity are inherently enforced thereby minimizing overhead which is important when sampling in NUMA machines. Furthermore, this design meshes well with certain PMU features such as the Precise-Event-Based

Sampling (PEBS) support of the Pentium 4 processor (see Section 5.4 for details).

4.3 Logical PMU

PMU register names and implementations are very diverse. On the Itanium processor architecture, they are implemented by actual PMC and PMD indirect registers. On the AMD Opteron [1] processors, they are called PERFSEL and PERFCTR indirect registers but are actually implemented by MSR registers. A portable tool would have to know about those names and the interface would have to change from one architecture to another to accommodate the names and types of the registers for the read and write operations. This would defeat our goal of having a uniform interface on all platforms.

To mask the diversity without compromising access to all PMU features, the interface exposes a *logical* PMU. This PMU is tailored to the underlying hardware PMU for properties such as the number of registers it implements. But it also guarantees the following properties across all architectures:

- the configuration registers are called PMC registers and are managed as 64-bit wide indirect registers
- the data registers are called PMD registers and are managed as 64-bit wide indirect registers
- counters are 64-bit wide unsigned integers

The mapping of PMC/PMD to actual PMU registers is defined by a PMU description table where each entry provides the default value, a bitmask of reserved fields, and the actual name of the register. The mapping is defined by the implementation and is accessible via a `sysfs` interface.

The routine to access the actual register is part of the architecture specific part of a perfmon2 implementation. For instance, on Itanium 2 processor, the mapping is defined such that the index in the table corresponds to the index of the actual PMU register, e.g., logical PMD0 corresponds to actual PMD0. The read function consists of a single `mov rXX=pmd[0]` instruction. On the Pentium M processor however, the mapping is defined as follows:

```
% cat /sys/kernel/perfmon/pmu_desc/mappings
PMC0:0x100000:0xffcfffff:PERFEVTSEL0
PMC1:0x100000:0xffcfffff:PERFEVTSEL1
PMD0:0x0:0xffffffffffffffff:PERFCTR0
PMD1:0x0:0xffffffffffffffff:PERFCTR1
```

When a tool writes to register `PMD0`, it writes to register `PERFEVTSEL0`. The actual register is implemented by MSR `0x186`. There is an architecture specific section of the PMU description table that provides the mapping to the MSR. The read function consist of a single `rdmsr` instruction.

On the Itanium 2 processors, we use this mapping mechanism to export the code (IBR) and data (DBR) debug registers as PMC registers because they can be used to restrict monitoring to a specific range of code or data respectively. There was no need to create an Itanium 2 processor specific system call in the interface to support this useful feature.

To make applications more portable, counters are always exposed as 64-bit wide unsigned integers. This is particularly interesting when sampling, see Section 5 for more details. Usually, PMUs implement narrower counters, e.g., 47 bits on Itanium 2 PMU, 40 bits on AMD Opteron PMU. If necessary, each implementation must emulate 64-bit counters. This can be accomplished fairly easily by leveraging the counter overflow interrupt capability present on all modern PMUs. Emulation can be turned off by applications on a per-counter basis, if necessary.

Oftentimes, it is interesting to associate PMU-based information with non-PMU based information such as an operating system resource or other hardware resource. For instance, one may want to include the time since monitoring has been started, the number of active networks connections, or the identification of the current process in a sample. The perfctr interface provides this kind of information, e.g., the virtual cycle counter, through a kernel data structure that is re-mapped to user level.

With perfmon2, it is possible to leverage the mapping table to define *Virtual PMD* registers, i.e., registers that do not map to actual PMU or PMU-related registers. This mechanism provides a uniform and extensible naming and access interface for those resources. Access to new resources can be added without breaking the ABI. When a tool invokes `pfm_read_pmds` on a virtual PMD register, a read call-back function, provided by the PMU description table, is invoked and returns a 64-bit value for the resource.

4.4 PMU description module

Hardware and software release cycles do not always align correctly. Although Linux kernel patches are produced daily on the `kernel.org` web site, most end-users really run packaged distributions which have a very different development cycle. Thus, new hardware may become available before there is an actual Linux distribution ready. Similarly, processors may be revised and new steppings may fix bugs in the PMU. Although providing updates is fairly easy nowadays, end-users tend to be reluctant to patch and recompile their own kernels.

It is important to understand that monitoring tool developers are not necessarily kernel developers. As such, it is important to provide simple mechanisms whereby they can enable early access to new hardware, add virtual PMD registers and run experimentations without full kernel patching and recompiling.

There are no technical reasons for having the PMU description tables built into the kernel. With a minimal framework, they can as well be implemented by kernel modules where they become easier to maintain. The perfmon2 interface provides a framework where a PMU description module can be dynamically inserted into the kernel at runtime. Only one module can be inserted at a time. When new hardware becomes available, assuming there is no changes needed in the architecture specific implementation, a new description module can be provided quickly. Similarly, it becomes easy to experiment with virtual PMD registers by modifying the description table and not the interface nor the core implementation.

5 Sampling Support

Statistical Sampling or *profiling* is the act of recording information about the execution of a program at some interval. The interval is commonly expressed in units of time, e.g., every 20ms. This is called Time-Based sampling (TBS). But the interval can also be expressed in terms of a number of occurrences of a PMU event, e.g., every 2000 L2 cache misses. This is called Event-Based sampling (EBS). TBS can easily be emulated with EBS by using an event with a fixed correlation to time, e.g., the number of elapsed cycles. Such emulation typically provides a much finer granularity than the operating system timer which is usually limited to millisecond at best. The interval, regardless of its unit, does not have to be constant.

At the end of an interval, the information is stored into a *sample* which may contain information as simple as where the thread was, i.e., the instruction pointer. It may also include values of some PMU registers or other hardware or software resources.

The quality of a profile depends mostly on the duration of the run and the number of samples collected. A good profile can provide a lot of useful information about the behavior of a program, in particular it can help identify bottlenecks. The difficulty is to manage to overhead involved with sampling. It is important to make sure that sampling does not perturb the execution of the monitored program such that it does not exhibit its normal behavior. As the sampling interval decreases, overhead increases.

The perfmon2 interface has an extensive set of features to support sampling. It is possible to manage sampling completely at the user level. But there is also kernel-level support to minimize the overhead. The interface provides support for EBS.

5.1 Sampling periods

All modern PMUs implement a counter overflow interrupt mechanism where the processor generates an interrupt whenever a counter wraps around to zero. Using this mechanism and supposing a 64-bit wide counter, it is possible to implement EBS by expressing a sampling period p as $2^{64} - p$ or in two's complement arithmetics as $-p$. After p occurrences, the counter overflows, an interrupt is generated indicating that a sample must be recorded.

Because all counters are 64-bit unsigned integers, tools do not have to worry about the actual width of counters when setting the period. When 64-bit emulation is needed, the implementation maintains a 64-bit software

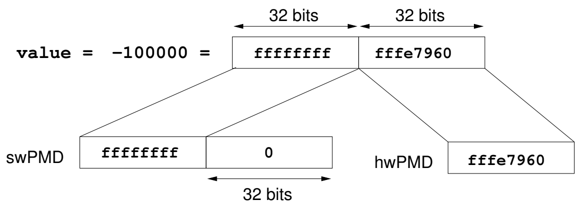


Figure 3: 64-bit counter emulation

value and loads only the low-order bits onto the actual register as shown in Figure 3. An EBS overflow is declared only when the 64-bit software-maintained value overflows.

The interface does not have the notion of a sampling period, all it knows about is PMD values. Thus a sampling period p, is programmed into a PMD by setting its value to $-p$. The number of sampling periods is only limited by the number of counters. Thus, it is possible to overlap sampling measurements to collect multiple profiles in one run.

For each counter, the interface provides three values which are used as follows:

- *value*: the value loaded into the PMD register when the context is attached. This is the initial value.
- *long_reset*: the value to reload into the PMD register after an overflow with user-level notification.
- *short_reset*: the value to reload into the PMD register after an overflow with no user-level notification.

The three values can be used to try and mask some of the overhead involved with sampling. The initial period would typically be large because it is not always interesting to capture samples in initialization code. The long and short reset values can be used to mask the *noise* generated by the PMU interrupt handler. We explain how they are used in Section 5.3.

5.2 Overflow notifications

To support sampling at the user level, it is necessary to inform the tool when a 64-bit overflow occurs. The notification can be requested per counter and is sent as a message. There is only one notification per interrupt even when multiple counters overflow at the same time.

Each perfmon context has a fixed-depth message queue. The fixed-size message contains information about the overflow such as which counter(s) overflowed, the instruction pointer, the current CPU at the time of the overflow. Each new message is appended to the queue which is managed as a FIFO.

Instead of re-inventing yet another notification mechanism, existing kernel interfaces are leveraged and messages are extracted using a simple `read` call on the file descriptor of the context. The benefit is that common interfaces such as `select` or `poll` can be used to wait on multiple contexts at the same time. Similarly, asynchronous notifications via `SIGIO` are also supported.

Regular file descriptor sharing semantic applies, thus it is possible to delegate notification processing to a specific thread or child process.

During a notification, monitoring is stopped. When monitoring another thread, it is possible to request that this thread be blocked while the notification is being processed. A tool may choose the block on notification option when the context is created. Depending on the type of sampling, it may be interesting to have the thread run just to keep the caches and TLB warm, for instance.

Once a notification is processed, the `pfm_restart` function is invoked. It is used to reset the overflowed counters using their *long* reset value, to resume monitoring, and potentially to unblock the monitored thread.

5.3 Kernel sampling buffer

It is quite expensive to send a notification to user level for each sample. This is particularly bad when monitoring another thread because there could be, at least, two context switches per overflow and a couple of system calls.

One way to minimize this cost, it is to *amortize* it over a large set of samples. The idea is to have the kernel directly record samples into a buffer. It is not possible to take page faults from the PMU interrupt handler, usually a high priority handler. As such the memory would have to be locked, an operation that is typically restricted to privileged users. As indicated earlier, sampling must be available to regular users, thus, the buffer is allocated by the kernel and marked as reserved to avoid being paged out.

When the buffer becomes full, the monitoring tool is notified. A similar approach is used by the `OProfile` and `VTUNE` interfaces. Several issues must be solved for the buffer to become useable:

- how to make the kernel buffer accessible to the user?
- how to reset the PMD values after an overflow when the monitoring tool is not involved?
- what format for the buffer?

The buffer can be made available via a `read` call. This is how `OProfile` and `VTUNE` work. Perfmon2 uses a different approach to try and minimize overhead. The buffer is re-mapped read-only into the user address space of the monitoring tool with a call to `mmap`, as shown in Figure 4. The content of the buffer is guaranteed consistent when a notification is received.

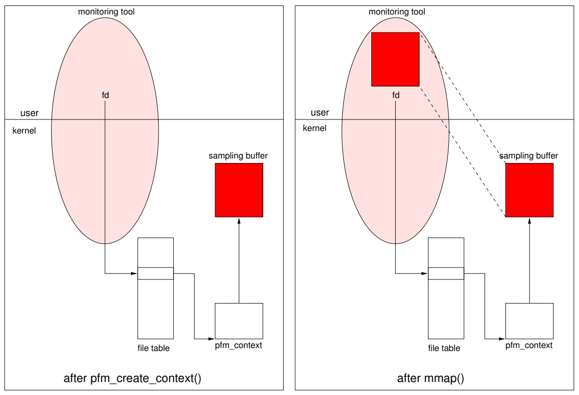


Figure 4: re-mapping the sampling buffer

On counter overflow, the kernel needs to know what value to reload into an overflowed PMD register. This information is passed, per register, during the `pfm_write_pmd` call. If the buffer does not become full, the kernel uses the *short* reset value to reload the counter.

When the buffer becomes full, monitoring is stopped and a notification is sent. Reset is deferred until the monitoring tool invokes `pfm_restart`, at which point, the buffer is marked as empty, the overflowed counter is reset with the *long* reset value and monitoring resumes.

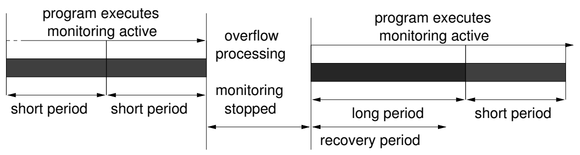


Figure 5: *short* vs. *long* reset values.

The distinction between *long* and *short* reset values allows tools to specify a different, potentially larger value, for the first period after an overflow notification. It is very likely that the user-level notification and subsequent processing will modify the CPU state, e.g., caches and TLB, such that when monitoring resumes, the execution will enter a *recovery* phase where its behavior may be different from what it would have been without monitoring. Depending on the type of sampling, the *long* vs. *short* reset values can be leveraged to hide that recovery period. This is demonstrated in Figure 5 which shows where the *long* reset value is used after overflow processing is completed. Of course, the impact and duration of the recovery period is very specific to each workload and CPU.

It is possible to request, per counter, that both reset values be randomized. This is very useful to avoid biased samples for certain measurements. The pseudo-random number generator does not need to be very fancy, simple variation are good enough. The randomization is specified by a seed value and a bitmask to limit the range of variation. For instance, a mask of `0xff` allows a variation in the interval `[0-255]` from the base value. The existing implementation uses the Carta [2] pseudo-random number generator because it is simple and very efficient.

A monitoring tool may want to record the values of certain PMD registers in each sample. Similarly, after each sample, a tool may want to reset certain PMD registers. This could be used to compute event deltas, for instance. Each PMD register has two bitmasks to convey this information to the kernel. Each bit in the bitmask represents a PMD register, e.g., bit 1 represents PMD1. Let us suppose that on overflow of PMD4, a tool needs to record PMD6 and PMD7 and then reset PMD7. In that case, the tool would initialize the sampling bitmask of PMD4 to `0xc0` and the reset bitmask to `0x80`.

With a kernel-level sampling buffer, the format in which samples are stored and what gets recorded becomes somehow fixed and it is more difficult to evolve. Monitoring tools can have very diverse needs. Some tools may want to store samples sequentially into the buffer, some may want to aggregate them immediately, others may want to record non PMU-based information, e.g., the kernel call stack.

As indicated earlier, it is important to ensure that existing interfaces such as `OProfile` or `VTUNE`, both using their own buffer formats, can be ported without having to modify a lot of their code. Similarly, It is important to ensure the interface can take advantage of advanced PMU support sampling such as the PEBS feature of the Pentium 4 processor.

Preserving a high level of flexibility for the buffer, while having it fully specified into the interface did not look very realistic. We realized that it would be very difficult to come up with a *universal* format that would satisfy all needs. Instead, the interface uses a radically different approach which is described in the next section.

5.4 Custom Sampling Buffer Formats

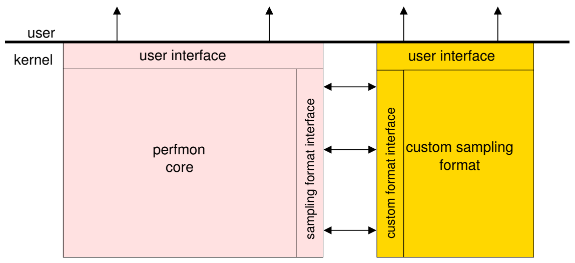


Figure 6: Custom sampling format architecture

The interface introduces a new flexible mechanism called *Custom Sampling Buffer Formats*, or formats for short. The idea is to remove the buffer format from the interface and instead provide a framework for extending the interface via specific sampling formats implemented by kernel modules. The architecture is shown in Figure 6.

Each format is uniquely identified by a 128-bit Universal Unique IDentifier (UUID) which can be generated by commands such as `uuidgen`. In order to use a format, a tool must pass this UUID when the context is created. It is possible to pass arguments, such as the buffer size, to a format when a context is created.

When the format module is inserted into the kernel, it registers with the perfmon core via a dedicated interface. Multiple formats can be registered. The list of available formats is accessible via a `sysfs` interface. Formats can also be dynamically removed like any other kernel module.

Each format provides a set of call-backs functions invoked by the perfmon core during certain operations. To make developing a format fairly easy, the perfmon core provides certain basic services such as memory allocation and the ability to re-map the buffer, if needed. Formats are not required to use those services. They may, instead, allocate their own buffer and expose it using a different interface, such as a driver interface.

At a minimum, a format must provide a call-back function invoked on 64-bit counter overflow, i.e., an interrupt handler. That handler does not bypass the core PMU interrupt handler which controls 64-bit counter emulation, overflow detection, notification, and monitoring masking. This layering make it very simple to write a handler. Each format controls:

- how samples are stored
- what gets recorded on overflow
- how the samples are exported to user-level
- when an overflow notification must be sent
- whether or not to reset counters after an overflow
- whether or not to mask monitoring after an overflow

The interface specifies a simple and relatively generic default sampling format that is built-in on all architectures. It stores samples sequentially in the buffer. Each sample has a fixed-size header containing information such as the instruction pointer at the time of the overflow, the process identification. It is followed by a variable-size body containing 64-bit PMD values stored in increasing index order. Those PMD values correspond to the information provided in the sampling bitmask of the overflowed PMD register. Buffer space is managed such that there can never be a partial sample. If multiple counters overflow at the same time, multiple contiguous samples are written.

Using the flexibility of formats, it was fairly easy to port the `OProfile` kernel code over to perfmon2. An new format was created to connect the perfmon2 PMU and `OProfile` interrupt handlers. The user-level `OProfile`, `opcontrol` tool was migrated over to use the perfmon2 interface to program the PMU. The resulting format is about 30 lines of C code. The OProfile buffer format and management kernel code was totally preserved.

Other formats have been developed since then. In particular we have released a format that implements *n*-way buffering. In this format, the buffer space is split into equal-size regions. Samples are stored in one region, when it fills up, the tool is notified but monitoring remains active and samples are stored in the next region. This idea is to limit the number of blind spots by never stopping monitoring on counter overflow.

The format mechanism proved particularly useful to implement support for the Pentium 4 processor Precise Event-Based Sampling (PEBS) feature where the CPU is directly writing samples to a designated region of memory. By having the CPU write the samples, the skew observed on the instruction pointer with typical interrupt-based sampling can be avoided, thus a much improved *precision* of the samples. That skew comes from the fact that the PMU interrupt is not generated exactly on the instruction where the counter overflowed. The phenomenon is especially important on deeply-pipelined processor implementations, such as Pentium 4 processor. With PEBS, there is a PMU interrupt when the memory region given to the CPU fills up.

The problem with PEBS is that the sample format is now fixed by the CPU and it cannot be changed. Furthermore, the format is different between the 32-bit and 64-bit implementations of the CPU. By leveraging the format infrastructure, we created two new formats, one for 32-bit and one for 64-bit PEBS with less than one hundred lines of C code each. Perfmon2 is the first to provide support for PEBS and it required no changes to the interface.

6 Event sets and multiplexing

On many PMU models, the number of counters is fairly limited yet certain measurements require lots of events. For instance, on the Itanium 2 processor, it takes about a dozen events to gather a cycle breakdown, showing how each CPU cycle is spent, yet there are only 4 counters. Thus, it is necessary to run the workload under test multiple times. This is not always very convenient as workloads sometimes cannot be stopped or are long to restart. Furthermore, this inevitably introduces fluctuations in the collected counts which may affect the accuracy of the results.

Even with a large number of counters, e.g., 18 for the Pentium 4 processor, there are still hardware constraints which make it difficult to collect some measurements in one run. For instance, it is fairly common to have constraints such as:

- event A and B cannot be measured together
- event A can only be measured on counter C.

Those constraints are unlikely to go away in the future because that could impact the performance of CPUs. An elegant solution to these problems is to introduce the notion of *events sets* where each set encapsulates the full PMU machine state. Multiple sets can be defined and they are multiplexed on the actual PMU hardware such that only one set if active at a time. At the end of the multiplexed run, the counts are scaled to compute an *estimate* of what they would have been, had they been collected for the entire duration of the measurement.

The accuracy of the scaled counts depends a lot of the switch frequency and the workload, the goal being to avoid blind spots where certain events are not visible because the set that measures them did not activate at the right time. The key point is to balance to the need for high switch frequency with higher overhead.

Sets and multiplexing can be implemented totally at the user level and this is done by the PAPI toolkit, for instance. However, it is critical to minimize the overhead especially for non self-monitoring measurements where it is extremely expensive to switch because it could incur, at least, two context switches and a bunch of system calls to save the current PMD values, reprogram the new PMC and PMD registers. During that window of time the monitored thread usually keeps on running opening up a large blind spot.

The perfmon2 interface supports events sets and multiplexing at the kernel level. Switching overhead is significantly minimized, blind spots are eliminated by the fact that switching systematically occurs in the context of the monitored thread.

Sets and multiplexing is supported for per-thread and system-wide monitoring and for both counting and sampling measurements.

6.1 Defining sets

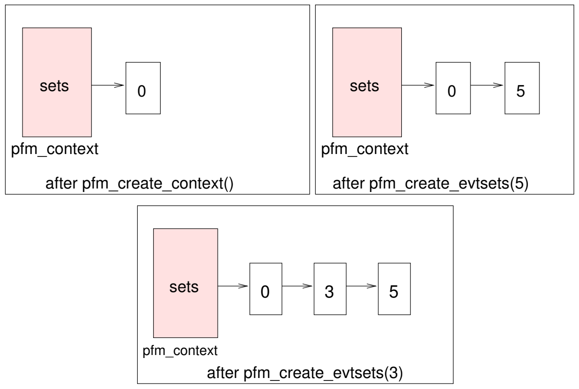

sets
0
pfm_context
after pfm_create_context()
sets
0
5
pfm_context
after pfm_create_evtsets(5)
sets
0
3
5
pfm_context
after pfm_create_evtsets(3)

Figure 7: creating sets.

Each context is created with a default event set, called *set0*. Sets can be dynamically created, modified, or deleted when the context is detached using the `pfm_create_evtsets` and `pfm_delete_evtsets` calls. Information, such as the number of activations of a set, can be retrieved with the `pfm_getinfo_evtsets` call. All these functions take array arguments and can, therefore, manipulate multiple sets per call.

A set is identified with a 16-bit number. As such, there is a theoretical limit of 65k sets. Sets are managed through an ordered list based on their identification numbers. Figure 7 shows the effect of adding *set5* and *set3* on the list.

Tools can program registers in each set by passing the set identification for each element of the array passed to the read or write calls. In one `pfm_write_pmcs` call it is possible to program registers for multiple sets.

6.2 Set switching

Set switching can be triggered by two different events: a timeout or a counter overflow. This is another innovation of the perfmon2 interface, again giving tools maximum flexibility. The type of *trigger* is determined, per set, when it is created.

The timeout is specified in micro-seconds when the set is created. The granularity of the timeout depends on the granularity of kernel internal timer tick, usually 1ms or 10ms. If the granularity is 10ms, then it is not possible to switch more than 100 times per second, i.e., the timeout cannot be smaller than 100μs. Because the granularity can greatly affect the accuracy of a measurement, the actual timeout, rounded up the the closest multiple of the timer tick, is returned by the `pfm_create_evtsets` call.

It is also possible to trigger a switch on counter overflow. To avoid dedicating a counter as a trigger, there is a trigger threshold value associated with each counter. At each overflow, the threshold value is decremented, when it reaches zero, switching occurs. It is possible to have multiple trigger counters per set, i.e., switch on multiple conditions.

The next set is determined by the position in the ordered list of sets. Switching is managed in a round-robin fashion. In the example from Figure 7, this means that the set following *set5* is *set0*.

Using overflow switching, it is possible to implement *counter cascading* where a counter starts counting only when a certain number of occurrences, n, of an event E is reached. In a first set, a PMC register is programmed to measure event E, the corresponding PMD register is initialized to $-n$, and its switch trigger is set to 1. The next set is setup to count the event of interest and it will activated only when there is an overflow in the first set.

6.3 Sampling

Sets are fully integrated with sampling. Set information is propagated wherever is necessary. The counter overflow notification carries the identification of the active set. The default sampling format fully supports sets. Samples from all sets are stored them into the same buffer. The active set at the time of the overflow is identified in the header of each sample.

7 Security

The interface is designed to be built into the base kernel, as such, it must follow the same security guidelines.

It is not possible to assume that tools will always be well-behaved. Each implementation must check arguments to calls. It must not be possible to use the interface for malicious attacks. A user cannot run a monitoring tool to extract information about a process or the system without proper permission.

All vector arguments have a maximum size to limit the amount of kernel memory necessary to perform the copy into kernel space. By nature, those calls are non-blocking and non-preemptible, ensuring that memory is eventually freed. The default limit is set to a page.

The sampling buffer size is also limited because it consumes kernel memory that cannot be paged out. There is a system-wide limit and a per-process limit. The latter is using the resource limit on locked memory (`RLIMIT_MEMLOCK`). The two-level protection is required to prevent users from launching lots of processes each allocating a small buffer.

In per-thread mode, the user credentials are checked against the permission of the thread to monitor when the context is attached. Typically, if a user cannot send a signal to the process, it is not possible to attach. By default, per-thread monitoring is available to all users, but a system administrator can limit to a user group. An identical, but separate, restriction is available for system-wide contexts.

On several architectures, such as Itanium, it is possible to read the PMD registers directly from user-level, i.e., with a simple instruction. There is always a provision to turn this feature off. The interface supports this mode of access by default for all self-monitoring per-thread context. It is turned off by default for all other configurations, thereby preventing spy applications from peeking at values left in PMD registers by others.

All size and user group limitations can be configured by a system administrator via a simple `sysfs` interface.

As for sampling, we are planning on adding a PMU interrupt throttling mechanism to prevent Denial-of-Service (DoS) attacks when applications set very high sampling rates.

8 Fast user-level PMD read

Invoking a system call to read a PMD register can be quite expensive compared to the cost of the actual instruction. On an Itanium 2 1.5GHz processor, for instance, it costs about 36 cycles to read a PMD with a single instruction and about 750 cycles via `pfm_read_pmds` which is not really optimized at this point. As a reference, the simplest system call, i.e., `getpid`, costs about 210 cycles.

On many PMU models, it is possible to directly read a PMD register from user level with a single instruction. This very lightweight mode of access is allowed by the interface for all self-monitoring threads. Yet, if actual counters width is less than 64-bit, only the partial value is returned. The software-maintained value requires a kernel call.

To enable fast 64-bit PMD read accesses from user level, the interface supports re-mapping of the software-maintained PMD values to user level for self-monitoring threads. This mechanism was introduced by the perfctr interface. This enables fast access on architectures without hardware support for direct access. For the others, this enables a full 64-bit value to be reconstructed by merging the high-order bits from the re-mapped PMD with the low-order bit obtained from hardware.

Re-mapping has to be requested when the context is created. For each event set, the PMD register values have to be explicitly re-mapped via a call to `mmap` on the file descriptor identifying the context. When a set is created a special *cookie* value is passed back by `pfm_create_evtset`. It is used as an offset for `mmap` and is required to identify the set to map. The mapping is limited to one page per set. For each set, the re-mapped region contains the 64-bit software value of each PMD register along with a status bit indicating whether the set is the active set or not. For non-active sets, the re-mapped value is the up-to-date full 64-bit value.

Given that the merge of the software and hardware values is not atomic, there can be a race condition if, for instance, the thread is preempted in the middle of building the 64-bit value. There is no way to avoid the race, instead the interface provides an atomic sequence number for each set. The number is updated each time the state of the set is modified. The number must be read by user-level code before and after reading the re-mapped PMD value. If the number is the same before and after, it means that the PMD value is current, otherwise the operation must be restarted. On the same Itanium 2 processor and without conflict, the cost is about 55 cycles to read the 64-bit value of a PMD register.

9 Status

A first generation of this interface has been implemented for the 2.6 kernel series for the Itanium Processor Family (IPF). It uses a single multiplexing system call, `perfmonctl`, and is missing events sets, PMU description tables, and fast user-level PMD reads. It is currently shipping with all major Linux distributions for this architecture.

The second generation interface, which we describe in this paper, currently exists as a kernel patch against the latest official 2.6 kernel from `kernel.org`. It supports the following processor architectures and/or models:

- all the Itanium processors
- the AMD Opteron processors in 64-bit mode
- the Intel Pentium M and P6 processors
- the Intel Pentium 4 and Xeon processors. That includes 32-bit and 64-bit (EM64T) processors. Hyper-Threading and PEBS are supported.
- the MIPS 5k and MIPS 20k processors
- preliminary support for IBM Power5 processor

Certain ports were contributed by other companies or developers. As our user community grows, we expect other contributions to both kernel and user-level code. The kernel patch has been widely distributed and has generated a lot of discussions on various Linux mailing lists.

Our goal is to establish perfmon2 as the standard Linux interface for hardware-based performance monitoring. We are in the process of getting it reviewed by the Community in preparation for a merge with the mainline kernel.

10 Existing tools

Several tools already exists for the interface. Most of them are only available for Itanium processors at this point, because an implementation exists since several years.

The first open-source tool to use the interface is *pfmon* [6] from HP Labs. This is a command-line oriented tool initially built to test the interface. It can collect counts and profiles on a per-thread or system-wide basis. It supports the Itanium, AMD Opteron, and Intel Pentium M processors. It is built on top of a helper library, called `libpfm`, which handles all the event encodings and assignment logic.

HP Labs also developed `q-tools` [7], a replacement program for `gprof`. Q-tools uses the interface to collect a flat profile and a statistical call graph of all processes running in a system. Unlike `gprof`, there is no need to recompile applications or the kernel. The profile and call graph include both user- and kernel-level execution. The tool only works on Itanium 2 processors because it leverages certain PMU features, in particular the Branch Trace Buffer. This tool takes advantage of the interface by overlapping two sampling measurements to collect the flat profile and call graph in one run.

The HP Caliper [5] is an official HP product which is free for non-commercial use. This is a professional tool which works with all major Linux distributions for Itanium processors. It collects counts or profiles on a per-thread or per-CPU basis. It is very simple to use and comes with a large choice of preset metrics such as flat profile (`fprof`), data cache misses (`dcache_miss`). It exploits all the advanced PMU features, such as the Branch Trace Buffer (BTB) and the Data Event Address Registers (D-EAR). The profiles are correlated to source and assembly code.

The PAPI toolkit has long been available on top of the perfmon2 interface for Itanium processors. We expect that PAPI will migrate over to perfmon2 on other architectures as well. This migration will likely simplify the code and allow better support for sampling and set multiplexing.

The BEA JRockit JVM on Linux/ia64, starting with version 1.4.2 is also exploiting the interface. The JIT compiler is using a, dynamically collected, per-thread profile to improve code generation. This technique [4], called Dynamic Profile Guided Optimization (DPGO), takes advantage of the efficient per-thread sampling support of the interface and of the ability of the Itanium 2 PMU to sample branches and locations of cache misses (Data Event Address Registers). What is particularly interesting about this example is that it introduces a new usage model. Monitoring is used each time a program runs and not just during the development phase. Optimizations are applied in the end-user environment and for the real workload.

11 Conclusion

We have designed the most advanced performance monitoring interface for Linux. It provides a uniform set of functionalities across all architectures making it easier to write portable performance tools. The feature set was carefully designed to allow efficient monitoring and a very high degree of flexibility to support a diversity of usage models and hardware architectures. The interface provides several key features such as custom sampling buffer formats, kernel support event sets multiplexing, and PMU description modules.

We have developed a multi-architecture implementation of this interface that support all major processors. On the Intel Pentium 4 processor, this implementation is the first to offer support for PEBS.

We are in the process of getting it merged into the mainline kernel. Several open-source and commercial tools are available on Itanium 2 processors at this point and we expect that others will be released for the other architectures as well.

Hardware-based performance monitoring is the key tool to understand how applications and operating systems behave. The monitoring information is used to drive performance improvements in applications, operating system kernels, compilers, and hardware. As processor implementation enhancements shift from pure clock speed to multi-core, multi-thread, the need for powerful monitoring will increase significantly. The perfmon2 interface is well suited to address those needs.

References

[1] AMD. *AMD64 Architecture Programmer's Manual: System Programming*, 2005. `http://www.amd.com/us-en/Processors/DevelopWithAMD`.

[2] David F. Carta. Two fast implementations of the minimal standard random number

generator. *Com. of the ACM*, 33(1):87–88, 1990. http://doi.acm.org/10.1145/76372.76379.

[3] Intel Coporation. The Itanium processor family architecture. http://developer.intel.com/design/itanium2/documentation.htm.

[4] Greg Eastman, Shirish Aundhe, Robert Knight, and Robert Kasten. Intel dynamic profile-guided optimization in the BEA JRockit^{TM} JVM. In *3rd Workshop on Managed Runtime Environments, MRE'05*, 2005. http://www.research.ibm.com/mre05/program.html.

[5] Hewlett-Packard Company. The Caliper performance analyzer. http://www.hp.com/go/caliper.

[6] Hewlett-Packard Laboratories. The pfmon tool and the libpfm library. http://perfmon2.sf.net/.

[7] Hewlett-Packard Laboratories. q-tools, and q-prof tools. http://www.hpl.hp.com/research/linux.

[8] Intel. *Intel Itanium 2 Processor Reference Manual for Software Development and Optimization*, April 2003. http://www.intel.com/design/itanium/documentation.htm.

[9] Intel. *IA-32 Intel Architecture Software Developers' Manual: System Programming Guide*, 2004. http://developer.intel.com/design/pentium4/manuals/index_new.htm.

[10] Intel Corp. The VTune^{TM} performance analyzer. http://www.intel.com/software/products/vtune/.

[11] Chi-Keung Luk and Robert Muth *et al.* Ispike: A post-link optimizer for the Intel Itanium architecture. In *Code Generation and Optimization Conference 2004 (CGO 2004)*, March 2004. http://www.cgo.org/cgo2004/papers/01_82_luk_ck.pdf.

[12] Mikael Pettersson. the Perfctr interface. http://user.it.uu.se/~mikpe/linux/perfctr/.

[13] Alex Shye *et al.* Analysis of path profiling information generated with performance monitoring hardware. In *INTERACT HPCA'04 workshop*, 2004. http://rogue.colorado.edu/draco/papers/interact05-pmu_pathprof.pdf.

[14] B.D̃ragovic *et al.* Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003. http://www.cl.cam.ac.uk/Research/SRG/netos/xen/architecture.html.

[15] J.Ãnderson *et al.* Continuous profiling: Where have all the cycles gone?, 1997. http://citeseer.ist.psu.edu/article/anderson97continuous.html.

[16] John Levon *et al.* Oprofile. http://oprofile.sf.net/.

[17] Robert Cohn *et al.* The PIN tool. http://rogue.colorado.edu/Pin/.

[18] Alex Tsariounov. The Prospect monitoring tool. http://prospect.sf.net/.

[19] University of Tenessee, Knoxville. Performance Application Programming Interface (PAPI) project. http://icl.cs.utk.edu/papi.

OCFS2: The Oracle Clustered File System, Version 2

Mark Fasheh
Oracle
mark.fasheh@oracle.com

Abstract

This talk will review the various components of the OCFS2 stack, with a focus on the file system and its clustering aspects. OCFS2 extends many local file system features to the cluster, some of the more interesting of which are posix unlink semantics, data consistency, shared readable mmap, etc.

In order to support these features, OCFS2 logically separates cluster access into multiple layers. An overview of the low level DLM layer will be given. The higher level file system locking will be described in detail, including a walkthrough of inode locking and messaging for various operations.

Caching and consistency strategies will be discussed. Metadata journaling is done on a per node basis with JBD. Our reasoning behind that choice will be described.

OCFS2 provides robust and performant recovery on node death. We will walk through the typical recovery process including journal replay, recovery of orphaned inodes, and recovery of cached metadata allocations.

Allocation areas in OCFS2 are broken up into groups which are arranged in self-optimizing "chains." The chain allocators allow OCFS2 to do fast searches for free space, and deallocation in a constant time algorithm. Detail on the layout and use of chain allocators will be given.

Disk space is broken up into clusters which can range in size from 4 kilobytes to 1 megabyte. File data is allocated in extents of clusters. This allows OCFS2 a large amount of flexibility in file allocation.

File metadata is allocated in blocks via a sub allocation mechanism. All block allocators in OCFS2 grow dynamically. Most notably, this allows OCFS2 to grow inode allocation on demand.

1 Design Principles

A small set of design principles has guided most of OCFS2 development. None of them are unique to OCFS2 development, and in fact, almost all are principles we learned from the Linux kernel community. They will, however, come up often in discussion of OCFS2 file system design, so it is worth covering them now.

1.1 Avoid Useless Abstraction Layers

Some file systems have implemented large abstraction layers, mostly to make themselves portable across kernels. The OCFS2 developers have held from the beginning that OCFS2 code would be Linux only. This has helped us in several ways. An obvious one is that it made the code much easier to read and navigate. Development has been faster because we can directly use the kernel features without worrying if another OS implements the same features, or worse, writing a generic version of them.

Unfortunately, this is all easier said than done. Clustering presents a problem set which most Linux file systems don't have to deal with. When an abstraction layer is required, three principles are adhered to:

- Mimic the kernel API.
- Keep the abstraction layer as thin as possible.
- If object life timing is required, try to use the VFS object life times.

1.2 Keep Operations Local

Bouncing file system data around a cluster can be very expensive. Changed metadata blocks, for example, must be synced out to disk before another node can read them. OCFS2 design attempts to break file system updates into node local operations as much as possible.

1.3 Copy Good Ideas

There is a wealth of open source file system implementations available today. Very often during OCFS2 development, the question "How do other file systems handle it?" comes up with respect to design problems. There is no reason to reinvent a feature if another piece of software already does it well. The OCFS2 developers thus far have had no problem getting inspiration from other Linux file systems.[1] In some cases, whole sections of code have been lifted, with proper citation, from other open source projects!

[1]Most notably Ext3.

2 Disk Layout

Near the top of the `ocfs2_fs.h` header, one will find this comment:

```
/*
 * An OCFS2 volume starts this way:
 * Sector 0: Valid ocfs1_vol_disk_hdr that cleanly
 * fails to mount OCFS.
 * Sector 1: Valid ocfs1_vol_label that cleanly
 * fails to mount OCFS.
 * Block 2: OCFS2 superblock.
 *
 * All other structures are found
 * from the superblock information.
 */
```

The OCFS disk headers are the only amount of backwards compatibility one will find within an OCFS2 volume. It is an otherwise brand new cluster file system. While the file system basics are complete, there are many features yet to be implemented. The goal of this paper then, is to provide a good explanation of where things are in OCFS2 today.

2.1 Inode Allocation Structure

The OCFS2 file system has two main allocation units, *blocks* and *clusters*. Blocks can be anywhere from 512 bytes to 4 kilobytes, whereas clusters range from 4 kilobytes up to one megabyte. To make the file system mathematics work properly, cluster size is always greater than or equal to block size. At format time, the disk is divided into as many cluster-sized units as will fit. Data is always allocated in clusters, whereas metadata is allocated in blocks

Inode data is represented in extents which are organized into a b-tree. In OCFS2, extents are represented by a triple called an *extent record*.

Extent records are stored in a large in-inode array which extends to the end of the inode block. When the extent array is full, the file system will allocate an *extent block* to hold the current

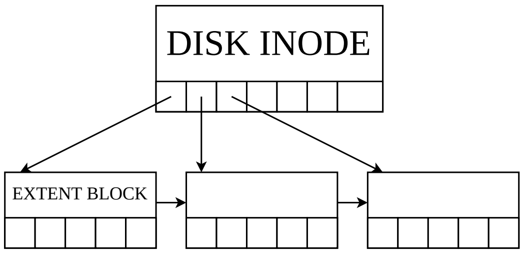


Figure 1: An Inode B-tree

Record Field	Field Size	Description
`e_cpos`	32 bits	Offset into the file, in clusters
`e_clusters`	32 bits	Clusters in this extent
`e_blkno`	64 bits	Physical disk offset

Table 1: OCFS2 extent record

array. The first extent record in the inode will be re-written to point to the newly allocated extent block. The `e_clusters` and `e_cpos` values will refer to the part of the tree underneath that extent. Bottom level extent blocks form a linked list so that queries accross a range can be done efficiently.

2.2 Directories

Directory layout in OCFS2 is very similar to Ext3, though unfortunately, htree has yet to be ported. The only difference in directory entry structure is that OCFS2 inode numbers are 64 bits wide. The rest of this section can be skipped by those already familiar with the dirent structure.

Directory inodes hold their data in the same manner which file inodes do. Directory data is arranged into an array of *directory entries*. Each directory entry holds a 64-bit inode pointer, a 16-bit record length, an 8-bit name length, an 8-bit file type enum (this allows us to avoid reading the inode block for type), and of course the set of characters which make up the file name.

2.3 The Super Block

The OCFS2 super block information is contained within an inode block. It contains a standard set of super block information—block size, compat/incompat/ro features, root inode pointer, etc. There are four values which are somewhat unique to OCFS2.

- `s_clustersize_bits` – Cluster size for the file system.
- `s_system_dir_blkno` – Pointer to the system directory.
- `s_max_slots` – Maximum number of simultaneous mounts.
- `s_first_cluster_group` – Block offset of first cluster group descriptor.

`s_clustersize_bits` is self-explanatory. The reason for the other three fields will be explained in the next few sections.

2.4 The System Directory

In OCFS2 file system metadata is contained within a set of *system files*. There are two types of system files, *global* and *node local*. All system files are linked into the file system via the hidden *system directory*[2] whose inode number is pointed to by the superblock. To find a system file, a node need only search the system directory for the name in question. The most common ones are read at mount time as a performance optimization. Linking to system files

[2]debugfs.ocfs2 can list the system dir with the `ls //` command.

from the system directory allows system file locatations to be completely dynamic. Adding new system files is as simple as linking them into the directory.

Global system files are generally accessible by any cluster node at any time, given that it has taken the proper cluster-wide locks. The `global_bitmap` is one such system file. There are many others.

Node local system files are said to be *owned* by a mounted node which occupies a unique *slot*. The maximum number of slots in a file system is determined by the `s_max_slots` superblock field. The `slot_map` global system file contains a flat array of node numbers which details which mounted node occupies which set of node local system files.

Ownership of a slot may mean a different thing to each node local system file. For some, it means that access to the system file is exclusive—no other node can ever access it. For others it simply means that the owning node gets preferential access—for an allocator file, this might mean the owning node is the only one allowed to allocate, while every node may delete.

A node local system file has its slot number encoded in the file name. For example, the journal used by the node occupying the third file system slot (slot numbers start at zero) has the name `journal:0002`.

2.5 Chain Allocators

OCFS2 allocates free disk space via a special set of files called *chain allocators*. Remember that OCFS2 allocates in clusters and blocks, so the generic term *allocation units* will be used here to signify either. The space itself is broken up into *allocation groups*, each of which contains a fixed number of allocation units. These groups are then *chained* together into a set of singly linked lists, which start at the allocator inode.

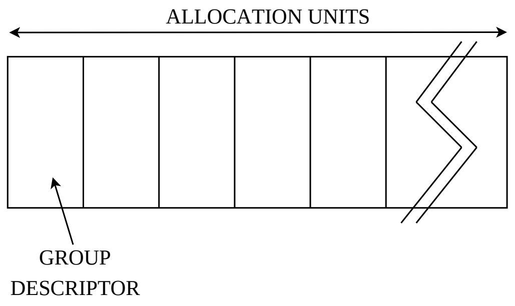


Figure 2: Allocation Group

The first block of the first allocation unit within a group contains an `ocfs2_group_descriptor`. The descriptor contains a small set of fields followed by a bitmap which extends to the end of the block. Each bit in the bitmap corresponds to an allocation unit within the group. The most important descriptor fields follow.

- `bg_free_bits_count` – number of unallocated units in this group.
- `bg_chain` – describes which group chain this descriptor is a part of.
- `bg_next_group` – points to the next group descriptor in the chain.
- `bg_parent_dinode` – pointer to disk inode of the allocator which owns this group.

Embedded in the allocator inode is an `ocfs2_chain_list` structure. The chain list contains some fields followed by an array of `ocfs2_chain_rec` records. An `ocfs2_chain_rec` is a triple which describes a chain.

- `c_blkno` – First allocation group.

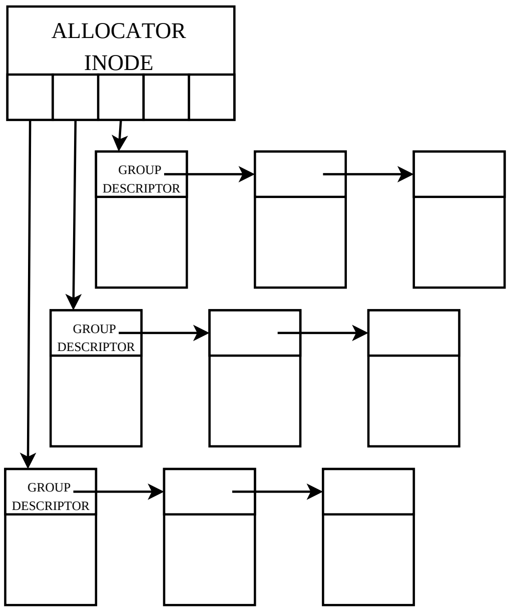


Figure 3: Chain Allocator

- `c_total` – Total allocation units.
- `c_free` – Free allocation units.

The two most interesting fields at the top of an `ocfs2_chain_list` are: `cl_cpg`, *clusters per group*; and `cl_bpc`, *bits per cluster*. The product of those two fields describes the total number of blocks occupied by each allocation group. As an example, the cluster allocator whose allocation units are clusters has a `cl_bpc` of 1 and `cl_cpg` is determined by `mkfs.ocfs2` (usually it just picks the largest value which will fit within a descriptor bitmap).

Chain searches are said to be self-optimizing. That is, while traversing a chain, the file system will re-link the group with the most number of free bits to the top of the list. This way, full groups can be pushed toward the end of the list and subsequent searches will require fewer disk reads.

2.6 Sub Allocators

The total number of file system clusters is largely static as determined by `mkfs.ocfs2` or optionally grown via `tunefs.ocfs2`. File system blocks however are dynamic. For example, an inode block allocator file can be grown as more files are created.

To grow a block allocator, `cl_bpc` clusters are allocated from the cluster allocator. The new `ocfs2_group_descriptor` record is populated and that block group is linked to the top of the smallest chain (wrapping back to the first chain if all are equally full). Other than the descriptor block, zeroing of the remaining blocks is skipped—when allocated, all file system blocks will be zeroed and written with a file system generation value. This allows `fsck.ocfs2` to determine which blocks in a group are valid metadata.

2.7 Local Alloc

Very early in the design of OCFS2 it was determined that a large amount of performance would be gained by reducing contention on the cluster allocator. Essentially the *local alloc* is a node local system file with an in-inode bitmap which caches clusters from the global cluster allocator. The local alloc file is **never** locked within the cluster—access to it is exclusive to a mounted node. This allows the block to remain valid in memory for the entire lifetime of a mount.

As the local alloc bitmap is exhausted of free space, an operation called a *window slide* is done. First, any unallocated bits left in the local alloc are freed back to the cluster allocator. Next, a large enough area of contiguous space is found with which to re-fill the local alloc. The cluster allocator bits are set, the local alloc bitmap is cleared, and the size and offset of the new window are recorded. If no suitable free space is found during the second step of a window slide, the local alloc is disabled for the remainder of that mount.

The size of the local alloc bitmap is tuned at mkfs time to be large enough so that most block group allocations will fit, but the total size would not be so large as to keep an inordinate amount of data unallocatable by other nodes.

2.8 Truncate Log

The *truncate log* is a node local system file with nearly the same properties of the local alloc file. The major difference is that the truncate log is involved in *de*-allocation of clusters. This in turn dictates a difference in disk structure.

Instead of a small bitmap covering a section of the cluster allocator, the truncate log contains an in-inode array of `ocfs2_truncate_rec` structures. Each `ocfs2_truncate_rec` is an extent, with a start (`t_start`) cluster and a length (`t_clusters`). This structure allows the truncate log to cover large parts of the cluster allocator.

All cluster de-allocation goes through the truncate log. It is flushed when full, two seconds after the most recent de-allocation, or on demand by a `sync(2)` call.

3 Metadata Consistency

A large amount of time is spent inside a cluster file system keeping metadata blocks consistent. A cluster file system not only has to track and journal dirty blocks, but it must understand which clean blocks in memory are still valid with respect to any disk changes which other nodes might initiate.

3.1 Journaling

Journal files in OCFS2 are stored as node local system files. Each node has exclusive access to its journal, and retains a cluster lock on it for the duration of its mount.

OCFS2 does block based journaling via the JBD subsystem which has been present in the Linux kernel for several years now. This is the same journaling system in use by the Ext3 file system. Documentation on the JBD disk format can be found online, and is beyond the scope of this document.

Though the OCFS2 team could have invented their own journaling subsystem (which could have included some extra cluster optimizations), JBD was chosen for one main reason—stability. JBD has been very well tested as a result of being in use in Ext3. For any journaled file system, stability in its journaling layer is critical. To have done our own journaling layer at the time, no matter how good, would have inevitably introduced a much larger time period of unforeseen stability and corruption issues which the OCFS2 team wished to avoid.

3.2 Clustered Uptodate

The small amount of code (less than 550 lines, including a large amount of comments) in `fs/ocfs2/updtodate.c` attempts to mimic the `buffer_head` caching API while maintaining those properties across the cluster.

The Clustered Uptodate code maintains a small set of metadata caching information on every OCFS2 memory inode structure (`struct ocfs2_inode_info`). The caching information consists of a single `sector_t` per block. These are stored in a 2 item array unioned with a red-black tree root item `struct rb_root`. If the number of buffers that require tracking grows larger than the array, then the red-black tree is used.

A few rules were taken into account before designing the Clustered Uptodate code:

1. All metadata changes are done under cluster lock.
2. All metadata changes are journaled.
3. All metadata reads are done under a read-only cluster lock.
4. Pinning `buffer_head` structures is not necessary to track their validity.
5. The act of acquiring a new cluster lock can flush metadata on other nodes and invalidate the inode caching items.

There are actually a very small number of exceptions to rule 2, but none of them require the Clustered Uptodate code and can be ignored for the sake of this discussion.

Rules 1 and 2 have the effect that the return code of `buffer_jbd()` can be relied upon to tell us that a `buffer_head` can be trusted. If it is in the journal, then we must have a cluster lock on it, and therefore, its contents are trustable.

Rule 4 follows from the logic that a newly allocated buffer head will not have its `BH_Uptodate` flag set. Thus one does not need to pin them for tracking purposes—a block number is sufficient.

Rule 5 instructs the Clustered Uptodate code to ignore `BH_Uptodate` buffers for which we do not have a tracking item—the kernel may think they're up to date with respect to disk, but the file system knows better.

From these rules, a very simple algorithm is implemented within `ocfs2_buffer_uptodate()`.

1. If `buffer_uptodate()` returns false, return false.
2. If `buffer_jbd()` returns true, return true.
3. If there is a tracking item for this block, return true.
4. Return false.

For existing blocks, tracking items are inserted after they are succesfully read from disk. Newly allocated blocks have an item inserted after they have been populated.

4 Cluster Locking

4.1 A Short DLM Tutorial

OCFS2 includes a DLM which exports a pared-down VMS style API. A full description of the DLM internals would require another paper the size of this one. This subsection will concentrate on a description of the important parts of the API.

A lockable object in the OCFS2 DLM is referred to as a *lock resource*. The DLM has no idea what is represented by that resource, nor does it care. It only requires a unique name by which to reference a given resource. In order to gain access to a resource, a process [3] acquires *locks* on it. There can be several locks on a resource at any given time. Each lock has a lock *level* which must be compatible with the levels of all other locks on the resource. All lock resources and locks are contained within a DLM *domain*.

[3]When we say *process* here, we mean a process which could reside on any node in the cluster.

Name	Access Type	Compatible Modes
EXMODE	Exclusive	NLMODE
PRMODE	Read Only	PRMODE, NLMODE
NLMODE	No Lock	EXMODE, PRMODE, NLMODE

Table 2: OCFS2 DLM lock Modes

In OCFS2, locks can have one of three levels, also known as lock *modes*. Table 2 describes each mode and its compatibility.

Most of the time, OCFS2 calls a single DLM function, `dlmlock()`. Via `dlmlock()` one can acquire a new lock, or *upconvert*, and *downconvert* existing locks.

```
typedef void (dlm_astlockfunc_t)(void *);

typedef void (dlm_bastlockfunc_t)(void *, int);

enum dlm_status dlmlock(
  struct dlm_ctxt *dlm,
  int mode,
  struct dlm_lockstatus *lksb,
  int flags,
  const char *name,
  dlm_astlockfunc_t *ast,
  void *data,
  dlm_bastlockfunc_t *bast);
```

Upconverting a lock asks the DLM to change its mode to a level greater than the currently granted one. For example, to make changes to an inode it was previously reading, the file system would want to upconvert its PRMODE lock to EXMODE. The currently granted level stays valid during an upconvert.

Downconverting a lock is the opposite of an upconvert—the caller wishes to switch to a mode that is more compatible with other modes. Often, this is done when the currently granted mode on a lock is incompatible with the mode another process wishes to acquire on its lock.

All locking operations in the OCFS2 DLM are asynchronous. Status notification is done via a set of callback functions provided in the arguments of a `dlmlock()` call. The two most important are the *AST* and *BAST* calls.

The DLM will call an AST function after a `dlmlock()` request has completed. If the status value on the `dlm_lockstatus` structure is DLM_NORMAL then the call has suceeded. Otherwise there was an error and it is up to the caller to decide what to do next.

The term BAST stands for *Blocking AST*. The BAST is the DLMs method of notifying the caller that a lock it is currently holding is blocking the request of another process.

As an example, if process A currently holds an EXMODE lock on resource *foo* and process B requests an PRMODE lock, process A will be sent a BAST call. Typically this will prompt process A to downconvert its lock held on *foo* to a compatible level (in this case, PRMODE or NLMODE), upon which an AST callback is triggered for both process A (to signify completion of the downconvert) and process B (to signify that its lock has been acquired).

The OCFS2 DLM supports a feature called *Lock Value Blocks*, or *LVBs* for short. An LVB is a fixed length byte array associated with a lock resource. The contents of the LVB are entirely up to the caller. There are strict rules to LVB access. Processes holding PRMODE and EXMODE locks are allowed to read the LVB value. Only processes holding EXMODE locks are allowed to write a new value to the LVB. Typically a read is done when acquiring or upconverting to a new PRMODE or EXMODE lock, while writes to the LVB are usually done when downconverting from an EXMODE lock.

4.2 DLM Glue

DLM glue (for lack of a better name) is a performance-critical section of code whose job it is to manage the relationship between the file system and the OCFS2 DLM. As such, DLM glue is the only part of the stack which knows about the internals of the DLM—regular file system code never calls the DLM API directly.

DLM glue defines several cluster lock types with different behaviors via a set of function pointers, much like the various VFS ops structures. Most lock types use the generic functions. The OCFS2 metadata lock defines most of its own operations for complexity reasons.

The most interesting callback that DLM glue requires is the *unblock* operation, which has the following definition:

int $(*$unblock$)($struct ocfs2_lock_res $*$, int $*)$;

When a blocking AST is recieved for an OCFS2 cluster lock, it is queued for processing on a per-mount worker thread called the *vote thread*. For each queued OCFS2 lock, the vote thread will call its `unblock()` function. If possible the `unblock()` function is to downconvert the lock to a compatible level. If a downconvert is impossible (for instance the lock may be in use), the function will return a non-zero value indicating the operation should be retried.

By design, the DLM glue layer **never** determines lifetiming of locks. That is dictated by the container object—in OCFS2, this is predominantly the `struct inode` which already has a set of lifetime rules to be obeyed.

Similarly, DLM glue is **only** concerned with multi-node locking. It is up to the callers to serialize themselves locally. Typically this is done via well-defined methods such as holding `inode->i_mutex`.

The most important feature of DLM glue is that it implements a technique known as *lock caching*. Lock caching allows the file system to skip costly DLM communication for very large numbers of operations. When a DLM lock is created in OCFS2 it is never destroyed until the container object's lifetime makes it useless to keep around. Instead, DLM glue maintains its current mode and instead of creating new locks, calling processes only take references on a single cached lock. This means that, aside from the initial acquisition of a lock and barring any BAST calls from another node, DLM glue can keep most lock / unlock operations down to a single integer increment.

DLM glue will not block locking processes in the case of an upconvert—say a `PRMODE` lock is already held, but a process wants exclusive access in the cluster. DLM glue will continue to allow processes to acquire `PRMODE` level references while upconverting to `EXMODE`. Similarly, in the case of a downconvert, processes requesting access at the target mode will not be blocked.

4.3 Inode Locks

A very good example of cluster locking in OCFS2 is the inode cluster locks. Each OCFS2 inode has three locks. They are described in locking order, outermost first.

1. `ip_rw_lockres` which serializes file read and write operations.
2. `ip_meta_lockres` which protects inode metadata.
3. `ip_data_lockres` which protects inode data.

The inode metadata locking code is responsible for keeping inode metadata consistent across the cluster. When a new lock is acquired at PRMODE or EXMODE, it is responsible for refreshing the struct inode contents. To do this, it stuffs the most common inode fields inside the lock LVB. This allows us to avoid a read from disk in some cases. The metadata unblock() method is responsible for waking up a checkpointing thread which forces journaled data to disk. OCFS2 keeps transaction sequence numbers on the inode to avoid checkpointing when unecessary. Once the checkpoint is complete, the lock can be downconverted.

The inode data lock has a similar responsibility for data pages. Complexity is much lower however. No extra work is done on acquiry of a new lock. It is only at downconvert that work is done. For a downconvert from EXMODE to PRMODE, the data pages are flushed to disk. Any downconvert to NLMODE truncates the pages and destroys their mapping.

OCFS2 has a cluster wide rename lock, for the same reason that the VFS has s_vfs_rename_mutex—certain combinations of rename(2) can cause deadlocks, even between multiple nodes. A comment in ocfs2_rename() is instructive:

```
/* Assume a directory hierarchy thusly:
 * a/b/c
 * a/d
 * a,b,c, and d are all directories.
 *
 * from cwd of 'a' on both nodes:
 * node1: mv b/c d
 * node2: mv d   b/c
 *
 * And that's why, just like the VFS, we need a
 * file system rename lock. */
```

Serializing operations such as mount(2) and umount(2) is the *super block lock*. File system membership changes occur only under an EXMODE lock on the super block. This is used to allow the mounting node to choose an appropriate slot in a race-free manner. The super block lock is also used during node messaging, as described in the next subsection.

4.4 Messaging

OCFS2 has a network *vote* mechanism which covers a small number of operations. The vote system stems from an older DLM design and is scheduled for final removal in the next major version of OCFS2. In the meantime it is worth reviewing.

Vote Type	Operation
OCFS2_VOTE_REQ_MOUNT	Mount notification
OCFS2_VOTE_REQ_UMOUNT	Unmount notification
OCFS2_VOTE_REQ_UNLINK	Remove a name
OCFS2_VOTE_REQ_RENAME	Remove a name
OCFS2_VOTE_REQ_DELETE	Query an inode wipe

Table 3: OCFS2 vote types

Each vote is broadcast to all mounted nodes (except the sending node) where they are processed. Typically vote messages about a given object are serialized by holding an EXMODE cluster lock on that object. That way the sending node knows it is the only one sending that exact vote. Other than errors, all votes except one return true. Membership is kept static during a vote by holding the super block lock. For mount/unmount that lock is held at EXMODE. All other votes keep a PRMODE lock. This way most votes can happen in parallel with respect to each other.

The mount/unmount votes instruct the other mounted OCFS2 nodes to the mount status of the sending node. This allows them in turn to track whom to send their own votes to.

The rename and unlink votes instruct receiving nodes to look up the dentry for the name being removed, and call the d_delete() function against it. This has the effect of removing the name from the system. If the vote is an unlink vote, the additional step of marking the inode as possibly orphaned is taken. The flag OCFS2_

`INODE_MAYBE_ORPHANED` will trigger additional processing in `ocfs2_drop_inode()`. This vote type is sent after all directory and inode locks for the operation have been acquired.

The delete vote is crucial to OCFS2 being able to support POSIX style unlink-while-open across the cluster. Delete votes are sent from `ocfs2_delete_inode()`, which is called on the last `iput()` of an orphaned inode. Receiving nodes simply check an *open count* on their inode. If the count is anything other than zero, they return a busy status. This way the sending node can determine whether an inode is ready to be truncated and deleted from disk.

5 Recovery

5.1 Heartbeat

The OCFS2 cluster stack heartbeats on disk and via its network connection to other nodes. This allows the cluster to maintain an idea of which nodes are alive at any given point in time. It is important to note that though they work closely together, the cluster stack is a separate entity from the OCFS2 file system.

Typically, OCFS2 disk heartbeat is done on every mounted volume in a contiguous set of sectors allocated to the `heartbeat` system file at file system create time. OCFS2 heartbeat actually knows nothing about the file system, and is only given a range of disk blocks to read and write. The system file is only used as a convenient method of reserving the space on a volume. Disk heartbeat is also never initiated by the file system, and always started by the `mount.ocfs2` program. Manual control of OCFS2 heartbeat is available via the `ocfs2_hb_ctl` program.

Each node in OCFS2 has a unique node number, which dictates which heartbeat sector it will periodically write a timestamp to. Optimizations are done so that the heartbeat thread only reads those sectors which belong to nodes which are defined in the cluster configuration. Heartbeat information from all disks is accumulated together to determine node liveness. A node need only write to one disk to be considered alive in the cluster.

Network heartbeat is done via a set of keep-alive messages that are sent to each node. In the event of a *split brain* scenario, where the network connection to a set of nodes is unexpectedly lost, a majority-based *quorum* algorithm is used. In the event of a 50/50 split, the group with the lowest node number is allowed to proceed.

In the OCFS2 cluster stack, disk heartbeat is considered the final arbiter of node liveness. Network connections are built up when a node begins writing to its heartbeat sector. Likewise network connections will be torn down when a node stops heartbeating to all disks.

At startup time, interested subsystems register with the heartbeat layer for *node up* and *node down* events. Priority can be assigned to callbacks and the file system always gets node death notification before the DLM. This is to ensure that the file system has the ability to mark itself needing recovery before DLM recovery can proceed. Otherwise, a race exists where DLM recovery might complete before the file system notification takes place. This could lead to the file system gaining locks on resources which are in need of recovery—for instance, metadata whose changes are still in the dead node's journal.

5.2 File System Recovery

Upon notification of an unexpected node death, OCFS2 will mark a *recovery bitmap*. Any file system locks which cover recoverable resources have a check in their locking path for any set bits in the recovery bitmap. Those paths will then block until the bitmap is clear again. Right now the only path requiring this check is the metadata locking code—it must wait on journal replay to continue.

A recovery thread is then launched which takes a `EXMODE` lock on the super block. This ensures that only one node will attempt to recover the dead node. Additionally, no other nodes will be allowed to mount while the lock is held. Once the lock is obtained, each node will check the `slot_map` system file to determine which journal the dead node was using. If the node number is not found in the slot map, then that means recovery of the node was completed by another cluster node.

If the node is still in the slot map then journal replay is done via the proper JBD calls. Once the journal is replayed, it is marked clean and the node is taken out of the slot map.

At this point, the most critical parts of OCFS2 recovery are complete. Copies are made of the dead node's truncate log and local alloc files, and clean ones are stamped in their place. A worker thread is queued to reclaim the disk space represented in those files, the node is removed from the recovery bitmap and the super block lock is dropped.

The last part of recovery—replay of the copied truncate log and local alloc files—is (appropriately) called *recovery completion*. It is allowed to take as long as necessary because locking operations are not blocked while it runs. Recovery completion is even allowed to block on recovery of other nodes which may die after its work is queued. These rules greatly simplify the code in that section.

One aspect of recovery completion which has not been covered yet is *orphan recovery*. The orphan recovery process must be run against the dead node's orphan directory, as well as the local orphan directory. The local orphan directory is recovered because the now dead node might have had open file descriptors against an inode which was locally orphaned—thus the `delete_inode()` code must be run again.

Orphan recovery is a fairly straightforward process which takes advantage of the existing inode life-timing code. The orphan directory in question is locked, and the recovery completion process calls `iget()` to obtain an inode reference on each orphan. As references are obtained, the orphans are arranged in a singly linked list. The orphan directory lock is dropped, and `iput()` is run against each orphan.

6 What's Been Missed!

Lots, unfortunately. The DLM has mostly been glossed over. The rest of the OCFS2 cluster stack has hardly been mentioned. The OCFS2 tool chain has some unique properties which would make an interesting paper. Readers interested in more information on OCFS2 are urged to explore the web page and mailing lists found in the references section. OCFS2 development is done in the open and when not busy, the OCFS2 developers love to answer questions about their project.

7 Acknowledgments

A huge thanks must go to all the authors of Ext3 from which we took much of our inspiration.

Also, without JBD OCFS2 would not be what it is today, so our thanks go to those involved in its development.

Of course, we must thank the Linux kernel community for being so kind as to accept our humble file system into their kernel. In particular, our thanks go to Christoph Hellwig and Andrew Morton whose guidance was critical in getting our file system code up to kernel standards.

8 References

The OCFS2 home page can be found at `http://oss.oracle.com/projects/ocfs2/`.

From there one can find mailing lists, documentation, and the source code repository.

tgt: Framework for Storage Target Drivers

Tomonori FUJITA
NTT Cyber Solutions Laboratories
`tomof@acm.org`

Mike Christie
Red Hat, Inc.
`michaelc@cs.wisc.edu`

Abstract

In order to provide block I/O services, Linux users have had to modify kernel code by hand, use binary kernel modules, or purchase specialized hardware. With the mainline kernel now having SCSI Parallel Interface (SPI), Fibre Channel (FC), iSCSI, and SCSI RDMA (SRP) initiator support, Linux target framework (tgt) aims to fill the gap in storage functionality by consolidating several target driver implementations and providing a SCSI protocol independent API that will simplify target driver creation and maintenance.

Tgt's key goal and its primary hurdle has been implementing a great portion of tgt in user space, while continuing to provide performance comparable to a target driver implemented entirely in the kernel. By pushing the SCSI state machine, I/O execution, and the management components of the framework outside of the kernel, it enjoys debugging, maintenance and mainline inclusion benefits. However, it has created new challenges. Both traditional kernel target implementations and tgt have had to transform Block Layer and SCSI Layer designs, which assume requests will be initiated from the top of the storage stack (the request queue's `make_request_fn()`) to an architecture that can efficiently handle asynchronous requests initiated by the the end of the stack (the low level drivers interrupt handler), but tgt also must efficiently communicate and synchronize with the user-space daemon that implements the SCSI target state machine and performs I/O.

1 Introduction

The SCSI protocol was originally designed to use a parallel bus interface and used to be tied closely to it. With the increasing demands of storage capacity and accessibility, it became obvious that Direct Attached Storage (DAS), the classic storage architecture, in which a host and storage devices are directly connected by system buses and parallel cable, cannot meet today's industry scalability and manageability requirements. This lead to the invention of Storage Area Network (SAN) technology, which enables hosts and storage devices to be connected via high-speed interconnection technologies such as Fibre Channel, Gigabit Ethernet, Infiniband, etc.

To enable SAN technology, the SCSI-3 architecture, as can be seen in Figure 1, brought an important change to the division of the standard into interface, protocol, device model, and command set. This allows device models and command sets with various transports (physical interfaces), such as Fibre Channel, Ethernet, and Infiniband. The device type specific command set, the primary command set, and transport are independent of each other.

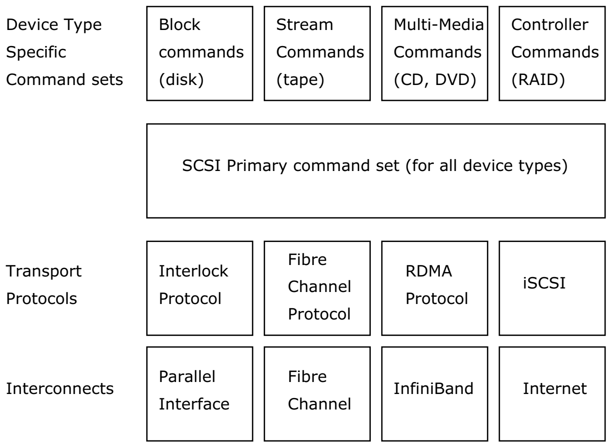


Figure 1: SCSI-3 architecture

1.1 What is a Target

SCSI uses a client-server model (Figure 2). Requests are initiated by a client, which in SCSI terminology is called an Initiator Device, and are processed by a server, which in SCSI terminology is known as a Target Device. Each target contains one or more logical units and provides services performed by device servers and task management functions performed by task managers. A logical unit is an object that implements one or more device functional models described in the SCSI command standards and processes commands (eq., reading from or writing to the media) [5].

Currently, the Linux kernel has support for several types of initiators including ones that use FC, TCP/IP, RDMA, or SPI for their transport protocol. There is however, no mainline target support.

2 Overview of Target Drivers

2.1 Target Driver

Generally in the past, a target driver is responsible for the following tasks:

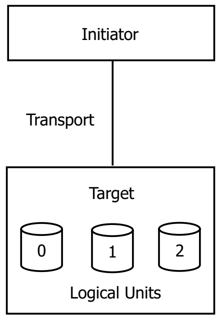


Figure 2: SCSI target and initiator

1. Handling its interconnect hardware interface and transport protocol.
2. Processing the primary and device specific command sets.
3. Accessing local devices (attached to the server directly) when necessary.

Since hardware interfaces are unique, the kernel needs a specific target driver for every hardware interface. However, the rest of the tasks are independent of hardware interfaces and transport protocols.

The duplication of code between tasks two and three lead to the necessity for a target framework that provides a API set useful for every target driver. In tgt, target drivers simply take SCSI commands from transport protocol packets, hand them over to the framework, and send back the responses to the clients via transport protocol packets. Figure 3 shows a simplified view of how hardware interfaces and transport protocols interact in tgt. It is more complicated than the above explanation of the ideal model due to some exceptions described below.

Tgt is integrated with Linux's SCSI Mid Layer (SCSI-ML), so it supports two hardware interface models:

Hardware A Host Bus Adapter (HBA) handles the major part of transport protocol

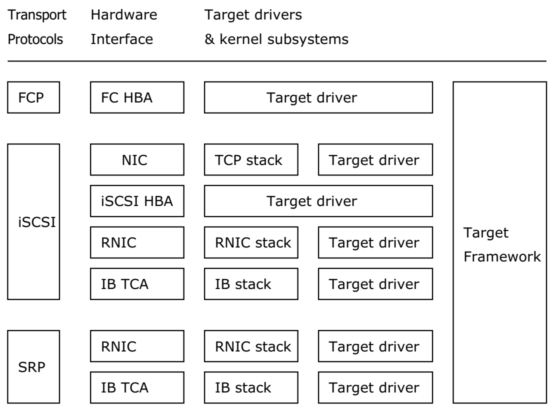


Figure 3: transport protocols and hardware interfaces

processing and the target driver implements the functionality to communicate between the HBA and tgt. Tgt needs a specific target driver for each type of HBA. FCP and SPI drivers follow this model. Drivers for other transports like iSCSI or SRP or for interconnects like iSER follow this model when there is specialized hardware to offload protocol or interconnect processing.

Software For transports like iSCSI and SRP or interconnects like iSER, a target driver can implement the transport protocol processing in a kernel module and access low level hardware through another subsystem such as the networking or infiniband stack. This allows a single target driver to work with various hardware interfaces.

3 Target Framework (tgt)

Our key design philosophy is implementing a significant portion of tgt in user space while maintaining performance comparable to a target driver implemented in kernel space. This conforms to the current trend of pushing code that can be implemented in user space out of

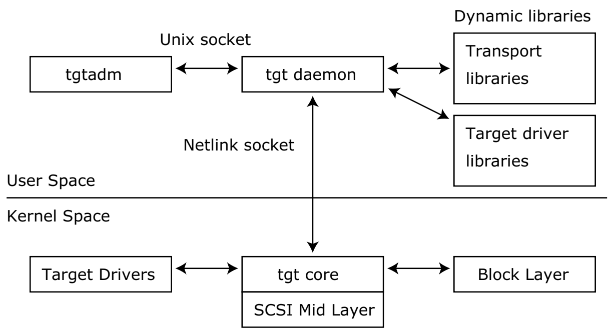


Figure 4: tgt components

the kernel [6] and enables developers to use rich user space libraries and development tools such as gdb.

As can be seen in Figure 4, the tgt architecture has two kernel components: the target driver and tgt core. The target driver's primary responsibilities are to manage the transport connections with initiator devices and pass commands and task management function requests between its hardware or interconnect subsystem and tgt core. tgt core is a simple connector between target drivers and the user space daemon (tgtd) that enables the driver to send tgtd a vector of commands or task management function requests through a netlink interface.

Tgt core was integrated into scsi-ml with minor modifications to the `scsi_host_template` and various scsi helper functions for allocating scsi commands. This allows tgt to rely on scsi-ml and the Block Layer for tricky issues such as hot-plugging, command buffer mapping, scatter gather list creation, and transport class integration. Note that tgt does not change the current scsi-ml API set, so normally the only modifications are required to the initiator low level driver's (LLD) interrupt handler to process target specific requests and to the transport classes so that they are able to present target specific attributes.

All SCSI protocol processing is performed in user space, so as can be seen by Figure 4 the bulk of the tgt is implemented in: tgtadm, tgtd, transport libraries and driver libraries. tgtadm is a simple management tool. A transport library is equivalent to a kernel transport class where functionality common to a set of drivers using the same transport can be placed. Driver libraries, are dynamically linked target driver specific libraries that can be used to implement functionality such as special setup and tear down operations. And, tgtd is the SCSI state machine that executes commands and task management requests.

The clear concern over the user space SCSI protocol processing is degraded performance[1]. We explain some techniques to overcome this problem as we discuss in more detail the tgt components.

3.1 API for Target Drivers

The target drivers interact with tgt core through a new tgt API set, and the existing mid-layer API set and data structures. For the most part, target drivers work in a very similar manner as the existing initiator drivers. In many cases the initiator only needs to implement the new target callbacks on the `scsi_host_template`: `transfer_response()`, `transfer_data()`, and `tsk_mgmt_response()`, to enable a target mode in its hardware. We examine the details of the new callbacks later in this section.

3.1.1 Kernel Setup

The first step in registering a target driver with scsi-ml and tgt core is to create a scsi host adapter instance. This is accomplished by calling the same functions that are used for the initiator: `scsi_host_alloc()` and `scsi_add_host()`. If an HBA will be running in both target and initiator mode then only a single call to each of those functions is necessary for each HBA. The final step in setting up a target driver is to allocate a `uspace_req_q` for each scsi host that will be running in target mode. A `uspace_req_q` is used by tgt core to send requests to user-space. It can be allocated and initialized by calling `scsi_tgt_alloc_queue()`.

[1]In the early days, tgt performed performance sensitive SCSI commands in kernel space (eq. read/write from/to storage devices). However, it turned out that the current design was able to achieve comparable performance.

3.1.2 Processing SCSI Commands in the Target Driver

The target driver needs to allocate the `scsi_cmnd` data structure for a SCSI command received from a client via `scsi_host_get_command()`. This corresponds to scsi-ml's `scsi_get_command()` usage for allocating a `scsi_cmnd` for each request coming from the Block Layer or scsi-ml Upper Layer Driver (ULD). While the former allocates the `scsi_cmnd` and the `request` data structures, the latter allocates only the `scsi_cmnd` data structure.

The target driver sets up and passes the `scsi_cmnd` data structure to tgt core via `scsi_tgt_queue_command()`. The following information is passed to tgt core from the target driver:

SCSI command buffer to contain SCSI command.

lun buffer buffer to represent logical unit number.

tag unique value to identify this SCSI command.

task attribute task attribute for ordering.

buffer length number of data bytes to transfer.

On completion of executing a SCSI command, tgt core invokes `transfer_response()`, which is specified in the `scsi_host_template` data structure.

`transfer_data()` is invoked prior to `transfer_response()` if a SCSI command involves data transfer. Like scsi-ml, a scatter gather list of pages at the `request_buffer` member in the `scsi_cmnd` data structure is used to specify data to transfer. Also like scsi-ml, tgt core utilizes Block Layer and scsi-ml helpers to create scatter gather lists within the `scsi_host_template` limits such as `max_sectors`, `dma_boundary`, `sg_tablesize`, and `use_clustering`.

If the SCSI command involves a target-to-initiator data transfer, a target driver transfers data pointed out by the scatter gather list to the client, and then invokes the function pointer passed as a argument of `transfer_data()` to notify tgt core of the completion of the operation.

If the SCSI command involves a initiator-to-target data transfer, the target driver copies (through a DMA operation or memcpy) data to the scatter gather list (the LLD or transport class requests the client to send data to write before the actual transfer if necessary), and then invokes the function pointer passed as a argument of `transfer_data()` to notify tgt core of the completion of the transfer.

Depending on the transfer size and hardware or transport limitations, tgt core may have to call `transfer_data()` multiple times to transmit the entire payload. To accomplish this, tgt is not able to easily reuse the existing Block Layer and SCSI API. This is due to tgt core executing from interrupt context, and because the scatter list APIs tgt utlizes were not intended for requests starting at end of the storage stack. To work around the Block Layer scatter gather list allocation function assumption that a request will normally be completed in one scatter list, tgt required two modifications or workarounds. The first and easiest, was the addition of an `offset` field to the `scsi_cmnd` to track where the LLD is currently at in the transfer. The more difficult change, and probably more of a hack, was for tgt core to maintain two lists of BIOs for each request. One list contains BIOs that have not been mapped to scatter lists and the second list contains BIOs that have been mapped into scatter gather lists, completed, and need to be unmapped from process context when the command is completed.

3.1.3 Task Management Function

A target driver can send task management function (TMF) requests to tgt core via `scsi_tgt_tsk_mgmt_request()`.

The first argument is the TMF type. Currently, the supported TMF types are `ABORT_TASK`, `ABORT_TASK_SET`, and `LOGICAL_UNIT_RESET`.

The second argument is the tag value to identify a command to abort. This corresponds to the tag argument of `scsi_tgt_queue_command()` and used only with `ABORT_TASK`.

The third argument is the lun buffer to identify a logical unit against which the TMF request is performed. This is used with TMF requests except for `ABORT_TASK`.

The last argument is a pointer to enable target drivers to identify this TMF request on completion of it.

tgt core invokes eh_abort_handler() per aborted command to allow the target driver to clean up any resources that it may have internally allocated for the command. Unlike when it is called by scsi-ml's error handler, the host is not guaranteed to be quiesced and may have initiator and target commands running.

Subsequently to eh_abort_handler(), tsk_mgmt_response() is invoked. The pointer to identify the completed TMF request is passed as the argument.

3.2 tgt core

tgt core conveys SCSI commands, TMF requests, and these results between target drivers and the user space daemon, tgtd, through a netlink interface, which enables a user space process to read and write a stream of data via the socket API. tgt core encapsulates the requests into netlink packets and sends them to user space to be executed. Then it receives netlink packets from user space, extracts the results of the operation, and performs auxiliary tasks in compliance with the results. Figure 5 shows the packet format for SCSI commands.

```
struct {
    int host_no;
    uint32_t cid;
    uint32_t data_len;
    uint8_t scb[16];
    uint8_t lun[8];
    int attribute;
    uint64_t tag;
} cmd_req;
```

Figure 5: Netlink packet for SCSI commands

Since moving large amounts of data via netlink leads to a performance drop because of the memory copies, for the command's data buffer tgt uses the memory mapped I/O technique utilized by the Linux SCSI generic (SG) device driver [4], which moves an address that the mmap() system call returns instead of lots of data.

When tgt core receives the address from user space, it increments the reference count on the pages of the mapped region and sets up the the scatter gather list in scsi_cmnd data structure. tgt core relies on the standard kernel API, bio_map_user(), scsi_alloc_sgtable(), and blk_rq_map_sg() for these chores. Similarly, bio_unmap_user() and scsi_free_sgtable() decrements the reference and cleans up the the scatter gather list. The former also marks the pages as dirty in case of initiator-to-target data transfer (WRITE_* command).

3.3 User Space Daemon (tgtd)

The user space daemon, tgtd, is the heart of tgt. It contains the SCSI state machine, executes requests and provides a consistent API for management via Unix domain sockets. It communicates with the target drivers through tgt core's netlink interface.

tgtd currently uses a single process model. This enables us to avoid tricky race conditions. Imagine that a SCSI command is sent to a particular device and a management request to remove the device comes at the same time. However, this means that tgtd always needs to work in an asynchronous manner.

The tgtd code is independent of transport protocols and target drivers. The transport-protocol dependent and target-driver dependent features, such as showing parameters, are implemented in dynamic libraries: transport-protocol libraries and target-driver libraries.

There are two instances that the administrators must understand: target and device. A target instance works as a SCSI target device server. Every working scsi host adapter that implements a target driver is bound to a particular target instance. Multiple scsi host adapter instances can be bound to a single target instance. A device instance corresponds to a SCSI logical unit. A target instance can have multiple device instances.

3.3.1 SCSI Command Processing in tgtd

In previous sections, the process by which a command is moved between the kernel and user space and how it is transferred between the target and initiator ports has been detailed. Now, the final piece of the process, where tgtd performs command execution, is described.

1. tgtd receives a netlink packet containing a SCSI command and finds the the target instance (the device instance is looked up if necessary) to which the command should be routed. As shown in Figure 5, the packet contains the host bus adapter ID and the logical unit buffer.
2. tgtd processes the task attribute to know when to execute the command (immediately or delay).
3. When the command is scheduled, tgtd executes it and sends the result to tgt core.
4. tgtd is notified via tgt core's netlink interface that the target driver has completed any needed data transfer and has successfully sent the response. tgtd is then able to free resources that it had allocated for the command.

In case of non-I/O commands, involving target-to-initiator data transfer, tgtd allocates buffer via `valloc()`, builds the response in it, and sends the address of the buffer to tgt core. The buffer is freed on completion of the command.

In case of I/O commands, tgtd maps the requested length starting at the offset from the device's file, and sends the address to the tgt core. On completion of the command, tgt calls the `munmap` system call.

To improve performance, if tgtd can map the whole device file (typically, it is possible with 64-bit architectures), tgtd does not call the `mmap` or `munmap` system calls per command. Instead, it maps the whole device file when the device instance is added to a target instance.

3.3.2 Task Management Function

When tgtd receives task management function requests to abort SCSI commands, it searches the commands, sends an abort request per the found commands, and then sends the TMF completion notification to tgt core.

Once tgt core marks pages as a dirty, it is impossible to stop them being committed to disk. Thus, tgtd does not try to abort a command if it is waiting for the completion. If tgtd receives a request to abort such command, it waits for the completion of the command and then sends the TMF completion notification indicating that the command is not found.

3.4 Configuration

The currently supported management operations are: creation and deletion of target and device instances and binding a host adapter instance to a target instance. All objects are independent of transport protocols. Transport-protocol dependent management requests (such as showing parameters) are performed by using the corresponding transport-protocol library.

The command-line management tool, *tgtadm*, is distributed together with tgt for ease of use, though tgtd provides a management API via Unix domain sockets so that administrators or vendors can implement their own management tools.

4 Status and the Future

Today, tgt implements only what is necessary to be able to benchmark it against kernel driver implementations and provide basic functionality. This has been due to the code being destabilized several times as a result of code review comments that have forced most of the code to be pushed to user space. This means that there is a long list of features to be implemented and ported from previous versions of tgt.

4.1 Target Driver Support

At the time of the writing of this paper, there is only one working target driver, ibmvstgt, which is a SRP target driver, though it works only for virtualization environments on IBM pSeries [2]. The virtual machines communicate via RDMA. One virtual machine (called the Virtual I/O server) works as a SRP server that provides I/O services for the rest of virtual machines. ibmvstgt is based on the IBM standalone (not a framework) target driver [3], ibmvscsis. By converting ibmvscsis to tgt more than 2,000 lines were removed from the original driver.

Currently, there is a Qlogic FC, qla2xxx-based, target driver being converted from kernel based target framework, SCST [9], to tgt. And, an Emulex FC, lpfc-based, target driver that utilized a GPL FC target framework is being worked on. Both of these require FC transport class Remote Port (rport) changes that will allow the FC class and tgt core to perform transport level recovery for the target LLDs.

The iSCSI code in mainline has also begun to be modified to support target mode. With the introduction of libiscsi a target could be implemented by creating a new iscsi transport module or by modifying the iscsi tcp module.

4.2 Kernel and User Space Communication

Netlink usage leads to memory allocations and memory copies for every netlink packet. It also suffers from frequent system calls. tgt avoids a significant performance drop by using the memory mapped I/O technique for the command data buffer, but there is still room for improvement. By removing the copy of the command and its status and sending a vector of commands or status we can reduce the memory copies and kernel user space trips.

Previous versions of tgt had used a mmapped packet socket (`AF_PACKET`), to send messages to user space. This removes netlink from the command execution initiation and proved to be a performance gain, but difficulties in the mmapped packet socket interface usage have prevented tgt from currently using it. Another option that provides high speed data transfers is the relayfs file system [10]. Unfortunately, both are unidirectional, and only communication from the kernel to user space is improved.

Another common technique for reducing system call usage is to send multiple requests in one invocation. During testing of tgt, this was attempted and showed promise, but was difficult to tune. Investigation will be resumed when more drivers are stabilized and can be benchmarked.

4.3 Virtualization

The logical unit that executes requests from a remote initiator is not limited to being backed by a local device that provides a SCSI interface. The object being used by tgt for the logical unit's device instance could be a IDE, SATA, SCSI, Device Mapper (DM), or Multiple Device (MD) device or a file. To provide this flexibility, previous versions of tgt provided *virtualized devices* for the clients regardless of the attached local object. tgtd had two types of virtualization:

device virtualization With device virtualization, a `device_type_handler` (DTH) emulates commands that cannot be executed directly by its device type. For example, a MD or DM device has no notion of a SCSI INQUIRY. In this case the DTH has the generic SCSI protocol emulation library execute the INQUIRY.

I/O virtualization With I/O virtualization a `io_type_handler` (IOTH) enables tgt to access regular and block device files using mmap or use specialized interfaces such as SG IO.

Many types and combinations of device and I/O virtualization are possible. A Virtual Tape Library (VTL) could provide a tape library using disk drives, or by using the virtual CD device and file I/O virtualization a tgt machine could provide cdrom devices for the client with ISO image files.

Another interesting virtualization mechanism is *passthrough*, directly passing SCSI commands to SCSI devices. This provides a feature, called storage bridge, to bind different SAN protocols. For example, suppose that you are already in a working FC environment, where there are is FC storage server and clients with a FC HBA. If you need to add new clients that need to access the FC storage server, however you cannot afford to buy new FC HBAs, an iSCSI-FC bridge can connect the existing FC network with a new iSCSI network.

Currently, tgt supports disk device virtualization and file I/O virtualization. The file I/O virtualization simply opens a file and accesses its data via the `mmap` system call.

5 Related Work

SCST is the first GPLed attempt to implement a complete framework for target drivers. There are several major differences between tgt and SCST: SCST is mature and contains many features, all the SCST components reside in kernel space, and SCST duplicates functionality found in scsi-ml and the Block Layer instead of exploiting and modifying those subsystems.

Besides ibmvscsis, there have been several standalone target drivers. iSCSI Enterprise Target software (IET) [1] is a popular iSCSI target driver for Ethernet adapters, which tgt has used as a base for istgt[2]. Other software iSCSI targets include the UNH and Intel implementations [8, 7] and there are several iSCSI and FC drivers for specific hardware like Qlogic and Chelsio.

[2]The first author has maintained IET. The failure to push it into the mainline kernel is one of the reasons why tgt was born.

6 Conclusion

This paper describes tgt, a new framework that adds storage target driver support to the SCSI subsystem. What differentiates tgt from other target frameworks and standalone drivers is its attempt to push the SCSI state model and I/O execution to user space.

By using the Block and SCSI layer, tgt has been able to quickly implement a solution that bypasses performance problems that result from executing memory copies to and from the kernel. However, the Block and SCSI Layers, were not designed to handle large asynchronous requests originating from the LLDs interrupt handlers. Since the Block Layer SG IO and SCSI Upper Layer Drivers like SG, share a common technique, code, and problems, we hope we will be able to find a final solution that will benefit tgt core and the rest of the kernel.

Tgt has undergone several rewrites as a result of code reviews, but is now reaching a point where hardware interface vendors and part time developers are collaborating to solidify tgt core and tgtd, implement new target drivers, make modifications to other kernel subsystems to support tgt, and implement new features.

The source code is available from `http://stgt.berlios.de/`

References

[1] iSCSI Enterprise Target software, 2004. `http://iscsitarget.sourceforge.net/`.

[2] Dave Boutcher and Dave Engebretsen. Linux Virtualization on IBM POWER5 Systems. In *Ottawa Linux Symposium*, pages 113–120, July 2004.

[3] Dave Boutcher. SCSI target for IBM Power5 LPAR, 2005. `http://patchwork.ozlabs.org/linuxppc64/patch?id=2285`.

[4] Douglas Gilbert. *The Linux SCSI Generic (sg) Driver*, 1999. `http://sg.torque.net/sg/`.

[5] T10 Technical Editor. SCSI architecture model-3, 2004. `http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf`.

[6] Edward Goggin, Alasdair Kergon, Christophe Varoqui, and David Olien. Linux Multipathing. In *Ottawa Linux Symposium*, pages 147–167, July 2005.

[7] Intel iSCSI Reference Implementation, 2001. `http://sourceforge.net/projects/intel-iscsi/`.

[8] UNH-iSCSI initiator and target, 2003. `http://unh-iscsi.sourceforge.net/`.

[9] Vladislav Bolkhovitin. Generic SCSI Target Middle Level for Linux, 2003. `http://scst.sourceforge.net/`.

[10] Karim Yaghmour, Robert Wisniewski, Richard Moore, and Michel Dagenais. relayfs: An efficient unified approach for trasmitting data from kernel to user space. In *Ottawa Linux Symposium*, pages 519–531, July 2003.

More Linux for Less

uClinux^{TM} on a $5.00 (US) Processor

Michael Hennerich
Analog Devices
hennerich@blackfin.uclinux.org

Robin Getz
Analog Devices
rgetz@blackfin.uclinux.org

Abstract

While many in the Linux community focus on enterprise and multi-processor servers, there are also many who are working and deploying Linux on the network edge. Due to its open nature, and the ability to swiftly develop complex applications, Linux is rapidly becoming the number one embedded operating system. However, there are many differences between running Linux on a Quad processor system with 16Gig of Memory and 250Gig of RAID storage than a on a system where the total cost of hardware is less than the price of a typical meal.

1 Introduction

In the past few years, Linux^{TM} has become an increasingly popular operating system choice not only in the PC and Server market, also in the development of embedded devices—particularly consumer products, telecommunications routers and switches, Internet appliances, and industrial and automotive applications.

The advantage of Embedded Linux is that it is a royalty-free, open source, compact solution that provides a strong foundation for an ever-growing base of applications to run on. Linux is a fully functional operating system (OS) with support for a variety of network and file handling protocols, a very important requirement in embedded systems because of the need to "connect and compute anywhere at anytime." Modular in nature, Linux is easy to slim down by removing utility programs, tools, and other system services that are not needed in the targeted embedded environment. The advantages for companies using Linux in embedded markets are faster time to market, flexibility, and reliability.

This paper attempts to answer several questions that all embedded developers ask:

- Why use a kernel at all?
- What advantages does Linux provide over other operating systems?
- What is the difference between Linux on x86 and low cost processors?
- Where can I get a kernel and how do I get started?
- Is Linux capable of providing real-time functionality?
- What are the possibilities to port a existing real-time application to a system running also Linux?

2 Why use a kernel at all

All applications require control code as support for the algorithms that are often thought of as the "real" program. The algorithms require data to be moved to and/or from peripherals, and many algorithms consist of more than one functional block. For some systems, this control code may be as simple as a "super loop" blindly processing data that arrives at a constant rate. However, as processors become more powerful, considerably more sophisticated control or signal processing may be needed to realize the processor's potential, to allow the processor to absorb the required functionality of previously supported chips, and to allow a single processor to do the work of many. The following sections provide an overview of some of the benefits of using a kernel on a processor.

2.1 Rapid Application Development

The use of the Linux kernel allows rapid development of applications compared to creating all of the control code required by hand. An application or algorithm can be created and debugged on an x86 PC using powerful desktop debugging tools, and using standard programming interfaces to device drivers. Moving this code base to an embedded linux kernel running on a low-cost embedded processor is trivial because the device driver model is exactly the same. Opening an audio device on the x86 Desktop is done in exactly the same was as on an embedded Linux system. This allows you to concentrate on the algorithms and the desired control flow rather than on the implementation details. Embedded Linux kernels and applications supports the use of C, C++, and assembly language, encouraging the development of code that is highly readable and maintainable, yet retaining the option of hand-optimizing if necessary.

2.2 Debugged Control Structures

Debugging a traditional hand-coded application can be laborious because development tools (compiler, assembler, and linker among others) are not aware of the architecture of the target application and the flow of control that results. Debugging complex applications is much easier when instantaneous snapshots of the system state and statistical runtime data are clearly presented by the tools. To help offset the difficulties in debugging software, embedded Linux kernels are tested with the same tests that many desktop distributions use before releasing a Linux kernel. This ensure that the embedded kernel is as bug-free as possible.

2.3 Code Reuse

Many programmers begin a new project by writing the infrastructure portions that transfers data to, from, and between algorithms. This necessary control logic usually is created from scratch by each design team and infrequently reused on subsequent projects. The Linux kernel provides much of this functionality in a standard, portable, and reusable manner. Furthermore, the kernel and its tight integration with the GNU development and debug tools are designed to promote good coding practice and organization by partitioning large applications into maintainable and comprehensible blocks. By isolating the functionality of subsystems, the kernel helps to prevent the morass all too commonly found in systems programming. The kernel is designed specifically to take advantage of commonality in user applications and to encourage code reuse. Each thread of execution is created from a user-defined template, either at boot time or dynamically by another thread. Multiple threads can be created from the same template, but the state associated with each created instance of the thread remains unique. Each thread template represents a complete encapsulation of an algorithm that is unaware of other threads in the system unless it has a direct dependency.

2.4 Hardware Abstraction

In addition to a structured model for algorithms, the Linux kernel provides a hardware abstraction layer. Presented programming interfaces allow you to write most of the application in a platform-independent, high-level language (C or C++). The Linux Application Programming Interface (API) is identical for all processors which support Linux, allowing code to be easily ported to a different processor core. When porting an application to a new platform, programmers must only address the areas necessarily specific to a particular processor—normally device drivers. The Linux architecture identifies a crisp boundary around these subsystems and supports the traditionally difficult development with a clear programming framework and code generation. Common devices can use the same driver interface (for example a serial port driver may be specific for a certain hardware, but the application $\longleftrightarrow$ serial port driver interface should be exactly the same, providing a well-defined hardware abstraction, and making application development faster).

2.5 Partitioning an Application

A Linux application or thread is an encapsulation of an algorithm and its associated data. When beginning a new project, use this notion of an application or thread to leverage the kernel architecture and to reduce the complexity of your system. Since many algorithms may be thought of as being composed of subalgorithm building blocks, an application can be partitioned into smaller functional units that can be individually coded and tested. These building blocks then become reusable components in more robust and scalable systems.

You define the behavior of Linux applications by creating the application. Many application or threads of the same type can be created, but for each thread type, only one copy of the code is linked into the executable code. Each application or thread has its own private set of variables defined for the thread type, its own stack, and its own C run-time context.

When partitioning an application into threads, identify portions of your design in which a similar algorithm is applied to multiple sets of data. These are, in general, good candidates for thread types. When data is present in the system in sequential blocks, only one instance of the thread type is required. If the same operation is performed on separate sets of data simultaneously, multiple threads of the same type can coexist and be scheduled for prioritized execution (based on when the results are needed).

2.6 Scheduling

The Linux kernel can be a preemptive multitasking kernel. Each application or thread begins execution at its entry point. Then, it either runs to completion or performs its primary function repeatedly in an infinite loop. It is the role of the scheduler to preempt execution of a an application or thread and to resume its execution when appropriate. Each application or thread is given a priority to assist the scheduler in determining precedence.

The scheduler gives processor time to the thread with the highest priority that is in the ready state. A thread is in the ready state when it is not waiting for any system resources it has requested.

2.7 Priorities

Each application or thread is assigned a dynamically modifiable priority. An application is limited to forty (40) priority levels. However, the number of threads at each priority is limited, in practice, only by system memory. Priority level one is the highest priority, and priority thirty is the lowest. The system maintains an idle thread that is set to a priority lower than that of the lowest user thread.

Assigning priorities is one of the most difficult tasks of designing a real-time preemptive system. Although there has been research in the area of rigorous algorithms for assigning priorities based on deadlines (for example, rate-monotonic scheduling), most systems are designed by considering the interrupts and signals triggering the execution, while balancing the deadlines imposed by the system's input and output streams.

2.8 Preemption

A running thread continues execution unless it requests a system resource using a kernel system call. When a thread requests a signal (semaphore, event, device flag, or message) and the signal is available, the thread resumes execution. If the signal is not available, the thread is removed from the ready queue; the thread is blocked. The kernel does not perform a context switch as long as the running thread maintains the highest priority in the ready queue, even if the thread frees a resource and enables other threads to move to the ready queue at the same or lower priority. A thread can also be interrupted. When an interrupt occurs, the kernel yields to the hardware interrupt controller. When the ISR completes, the highest priority thread resumes execution.

2.9 Application and Hardware Interaction

Applications should have minimal knowledge of hardware; rather, they should use device drivers for hardware control. A application can control and interact with a device in a portable and hardware abstracted manner through a standard set of APIs.

The Linux Interrupt Service Routine framework encourages you to remove specific knowledge of hardware from the algorithms encapsulated in threads. Interrupts relay information to threads through signals to device drivers or directly to threads. Using signals to connect hardware to the algorithms allows the kernel to schedule threads based on asynchronous events. The Linux run-time environment can be thought of as a bridge between two domains, the thread domain and the interrupt domain. The interrupt domain services the hardware with minimal knowledge of the algorithms, and the thread domain is abstracted from the details of the hardware. Device drivers and signals bridge the two domains.

2.10 Downside of using a kernel

- Memory consumption: to have a usable Linux system, you should consider having at least 4–8 MB of SDRAM, and at least 2MB of Flash.
- Boot Time: the kernel is fast, but sometimes not fast enough, expect to have a 2–5 second boot time.
- Interrupt Latency: On occasions, a Linux device driver, or even the kernel, will disable interrupts. Some critical kernel operations can not be interrupted, and it is unfortunate, but interrupts must be turned off for a bit. Care has been taken to keep critical regions as short as possible as they

cause increased and variable interrupt latency.

- Robustness: although a kernel has gone through lots of testing, and many people are using it, it is always possible that there are some undiscovered issues. Only you can test it in the configuration that you will ship it.

3 Advantages of Linux

Despite the fact that Linux was not originally designed for use in embedded systems, it has found its way into many embedded devices. Since the release of kernel version 2.0.x and the appearance of commercial support for Linux on embedded processors, there has been an explosion of embedded devices that use Linux as their OS. Almost every day there seems to be a new device or gadget that uses Linux as its operating system, in most cases going completely unnoticed by the end users. Today a large number of the available broadband routers, firewalls, access points, and even some DVD players utilize Linux, for more examples see Linuxdevices.[1]

Linux offers a huge amount of drivers for all sorts of hardware and protocols. Combine that with the fact that Linux does not have run-time royalties, and it quickly becomes clear why there are so many developers using Linux for their devices. In fact, in a recent embedded survey, 75% of developers indicated they are using, or are planning on using an open source operating system.[2]

[1] `http://www.linuxdevices.org`

[2] Embedded systems survey `http://www.embedded.com/showArticle.jhtml?articleID=163700590`

Many commercial and non-commercial Linux kernel trees and distributions enable a wide varity of choices for the embedded developer.

One of the special trees is the uClinux (Pronounced *you-see-linux*, the name uClinux comes from combining the greek letter *mu* (μ) and the English capital *C*. *Mu* stands for *micro*, and the *C* is for *controller*) kernel tree, at `http://www.uclinux.org`. This is a distribution which includes a Linux kernel optimized for low-cost processors, including processors without a Memory Management Unit (MMU). While the nommu kernel patch has been included in the official Linux 2.6.x kernel, the most up-to-date development activity and projects can be found at uClinux Project Page and Blackfin/uClinux Project Page[3]. Patches such as these are used by commercial Linux vendors in conjunction with their additional enhancements, development tools, and documentation to provide their customers an easy-to-use development environment for rapidly creating powerful applications on uClinux.

Contrary to most people's understanding, uClinux is not a "special" Linux kernel tree, but the name of a distribution, which goes through testing on low-cost embedded platforms.

`www.uclinux.org` provides developers with a Linux distribution that includes different kernels (2.0.x, 2.4.x, 2.6.x) along with required libraries; basic Linux shells and tools; and a wide range of additional programs such as web server, audio player, programming languages, and a graphical configuration tool. There are also programs specially designed with size and efficiency as their primary considerations. One example is busybox, a multicall binary, which is a program that includes the functionality of a lot of smaller programs and acts like any one of them if it is called by the appropriate name. If busybox is linked to `ls` and contains the `ls`

[3] `http://www.blackfin.uclinux.org`

code, it acts like the `ls` command. The benefit of this is that busybox saves some overhead for unique binaries, and those small modules can share common code.

In general, the uClinux distribution is more than adequate enough to compile a full Linux image for a communication device, like a router, without writing a single line of code.

4 Differences between MMU Linux and noMMU Linux

Since Linux on processors with MMU and without MMU are similar to UNIX256 in that it is a multiuser, multitasking OS, the kernel has to take special precautions to assure the proper and safe operation of up to thousands of processes from different users on the same system at once. The UNIX security model, after which Linux is designed, protects every process in its own environment with its own private address space. Every process is also protected from processes being invoked by different users. Additionally, a Virtual Memory (VM) system has additional requirements that the Memory Management Unit (MMU) must handle, like dynamic allocation of memory and mapping of arbitrary memory regions into the private process memory.

Some processors, like Blackfin, do not provide a full-fledged MMU. These processors are more power efficient and significantly cheaper than the alternatives, while sometimes having higher performance.

Even on processors featuring Virtual Memory, some system developers target their application to run without the MMU turned on, because noMMU Linux can be significantly faster than Linux on the same processor. Overhead of MMU operations can be significant. Even when a MMU is available, it is sometimes not used in systems with high real-time constraints. Context switching and Inter Process Communication (IPC) can also be several times faster on uClinux. A benchmark on an ARM9 processor, done by H.S. Choi and H.C. Yun, has proven this.[4]

To support Linux on processors without an MMU, a few trade-offs have to be made:

1. No real memory protection (a faulty process can bring the complete system down)
2. No fork system call
3. Only simple memory allocation
4. Some other minor differences

4.1 Memory Protection

Memory protection is not a real problem for most embedded devices. Linux is a very stable platform, particularly in embedded devices, where software crashes are rarely observed. Even on a MMU-based system running Linux, software bugs in the kernel space can crash the whole system. Since Blackfin has memory protection, but not Virtual Memory, Blackfin/uClinux has better protection than other noMMU systems, and does provide some protection from applications writing into peripherals, and therefore will be more robust than uClinux running on different processors.

There are two most common principal reasons causing uClinux to crash:

- Stack overflow: When Linux is running on an architecture where a full MMU exists, the MMU provides Linux programs

[4]`http://opensrc.sec.samsung.com/document/uc-linux-04_sait.pdf`

basically unlimited stack and heap space. This is done by the virtualization of physical memory. However most embedded Linux systems will have a fixed amount of SDRAM, and no swap, so it is not really "unlimited." A program with a memory leak can still crash the entire system on embedded Linux with a MMU and virtual memory.

Because noMMU Linux can not support VM, it allocates stack space during compile time at the end of the data for the executable. If the stack grows too large on noMMU Linux, it will overwrite the static data and code areas. This means that the developer, who previously was oblivious to stack usage within the application, must now be aware of the stack requirements.

On gcc for Blackfin, there is a compiler option to enable stack checking. If the option `-fstack-limit-symbol=_stack_start` is set, the compiler will add in extra code which checks to ensure that the stack is not exceeded. This will ensure that random crashes due to stack corruption/overflow will not happen on Blackfin/uClinux. Once an application compiled with this option and exceeds its stack limit, it gracefully dies. The developer then can increase the stack size at compile time or with the `flthdr` utility program during runtime. On production systems, stack checking can either be removed (increase performance/reduce code size), or left in for the increase in robustness.

- Null pointer reference: The Blackfin MMU does provide partial memory protection, and can segment user space from kernel (supervisor) space. On Blackfin/uClinux, the first 4K of memory starting at NULL is reserved as a buffer for bad pointer dereferences. If an application uses a uninitialized pointer that reads or writes into the first 4K of memory, the application will halt. This will ensure that random crashes due to uninitialized pointers are less likely to happen. Other implementations of noMMU Linux will start writing over the kernel.

4.2 No Fork

The second point can be little more problematic. In software written for UNIX or Linux, developers sometimes use the fork system call when they want to do things in parallel. The `fork()` call makes an exact copy of the original process and executes it simultaneously. To do that efficiently, it uses the MMU to map the memory from the parent process to the child and copies only those memory parts to that child it writes. Therefore, uClinux cannot provide the `fork()` system call. It does, however, provide `vfork()`, a special version of `fork()`, in which the parent is halted while the child executes. Therefore, software that uses the `fork()` system call has to be modified to use either `vfork()` or POSIX threads that uClinux supports, because they share the same memory space, including the stack.

4.3 Memory Allocation

As for point number three, there usually is no problem with the malloc support noMMU Linux provides, but sometimes minor modifications may have to be made. Memory allocation on uClinux can be very fast, but on the other hand a process can allocate all available memory. Since memory can be only allocated in contiguous chunks, memory fragmentation can be sometimes an issue.

4.4 Minor Differences

Most of the software available for Linux or UNIX (a collection of software can be found on `http://freshmeat.net`) can be directly compiled on uClinux. For the rest, there is usually only some minor porting or tweaking to do. There are only very few applications that do not work on uClinux, with most of those being irrelevant for embedded applications.

5 Developing with uClinux

When selecting development hardware, developers should not only carefully make their selection with price and availability considerations in mind, but also look for readily available open source drivers and documentation, as well as development tools that makes life easier—e.g., kernel, driver and application debugger, profiler, strace.

/subsectionTesting uClinux Especially when developing with open source—where software is given as-is—developers making a platform decision should also carefully have a eye on the test methodology for kernel, drivers, libraries, and toolchain. After all, how can a developer, in a short time, determine if the Linux kernel running on processor A is better or worse than running on processor B?

The simplest way to test a new kernel on a new processor is to just boot the platform, and try out the software you normally run. This is an important test, because it tests most quickly the things that matter most, and you are most likely to notice things that are out of the ordinary from the normal way of working. However, this approach does not give widespread test coverage; each user tends to use the GNU/Linux system only for a very limited range of the available functions it offers and it can take significant time to build the processor tool chain, build the kernel, and download it to the target for the testing.

Another alternative is to run test suites. These are software packages written for the express purpose of testing, and they are written to cover a wide range of functions and often to expose things that are likely to go wrong.

The Linux Test Project (LTP), as an example, is a joint project started by SGI and maintained by IBM, that has a goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux. The LTP test suite contains a collection of tools for testing the Linux kernel and related features. Analog Devices, Inc., sponsored the porting of LTP to architectures supported by noMMU Linux.

Testing with test suites applies not only the kernel, also all other tools involved during the development process. If you can not trust your compiler or debugger, then you are lost. Blackfin/uClinux uses DejaGnu to ease and automate the over 44,000 toolchain tests, and checking of their expected results while running on target hardware. In addition there are test suites included in Blackfin/uClinux to do automated stress tests on kernel and device drivers using expect scripts. All these tests can be easily reproduced because they are well documented.

Here are the test results for the Blackfin gcc-4.x compiler.

```
=== gas Summary ===
# of expected passes               79

== binutils Summary ===
# of expected passes               26
# of untested testcases             7

=== gdb Summary ===
# of expected passes             9018
# of unexpected failures           62
# of expected failures             41
# of known failures                27
# of unresolved testcases           9
# of untested testcases             5
# of unsupported tests             32

=== gcc Summary ===
# of expected passes            36735
# of unexpected failures           33
# of unexpected successes           1
# of expected failures             75
# of unresolved testcases          28
# of untested testcases            28
# of unsupported tests            393

=== g++ Summary ===
# of expected passes            11792
# of unexpected failures           10
# of unexpected successes           1
# of expected failures             67
# of unresolved testcases          14
# of unsupported tests            165
```

All of the unexpected failures have been analysed to ensure that the toolchain is as stable as possible with all types of software that someone could use it with.

6 Where can I get uClinux and how do I get started?

Normally, the first selection that is made once Linux is chosen as the embedded operating system, is to identify the lowest cost processor that will meet the performance targets. Luckily, many silicon manufacturers are fighting for this position.

During this phase of development it is about the 5 processor Ps.

- Penguins
- Price
- Power
- Performance
- Peripherals

6.1 Low-cost Processors

Although the Linux kernel supports many architectures, including alpha, arm, frv, h8300, i386, ia64, m32r, m68k, mips, parisc, powerpc, s390, sh, sparc, um, v850, x86_64, and xtensa, many Linux developers are surprised to hear of a recent new Linux port to the Blackfin Processor.

Blackin Processors combine the ability for real-time signal processing and the functionality of microprocessors, fulfilling the requirements of digital audio and communication applications. The combination of a signal processing core with traditional processor architecture on a single chip avoids the restrictions, complexity, and higher costs of traditional heterogeneous multiprocessor systems.

All Blackfin Processors combine a state-of-the-art signal processing engine with the advantages of a clean, orthogonal RISC-like microprocessor instruction set and Single Instruction Multiple Data (SIMD) multimedia capabilities into a single instruction set architecture. The Micro Signal Architecture (MSA) core is a dual-MAC (Multiply Accumulator Unit) modified Harvard Architecture that has been designed to have unparalleled performance on typical signal processing[5] algorithms, as well as standard program flow and arbitrary bit manipulation operations mainly used by an OS.

The single-core Blackfin Processors have two large blocks of on-chip memory providing high bandwidth access to the core. These memory blocks are accessed at full processor core speed (up to 756MHz). The two memory blocks sitting next to the core, referred to as L1 memory, can be configured either as data or instruction SRAM or cache. When configured as cache, the speed of executing external code from SDRAM is nearly on par with running the code from internal memory. This feature is especially well suited for running the uClinux kernel, which doesn't fit into internal memory. Also, when programming in C, the memory access optimization can be left up to the core by using cache.

6.2 Development Environment

A typical uClinux development environment consists of a low-cost Blackfin STAMP board, and the GNU Compiler Collection (gcc cross compiler) and the binutils (linker, assembler, etc.) for the Blackfin Processor. Additionally, some GNU tools like `awk`, `sed`, `make`, `bash`, etc., plus `tcl/tk` are needed, although they usually come by default with the desktop Linux distribution.

An overview of some of the STAMP board features are given below:

- ADSP-BF537 Blackfin device with JTAG interface
- 500MHz core clock
- Up to 133MHz system clock
- 32M x 16bit external SDRAM (64MB)
- 2M x 16bit external flash (4MB)
- 10/100 Mbps Ethernet Interface (via on-chip MAC, connected via DMA)
- CAN Interface
- RS-232 UART interface with DB9 serial connector
- JTAG ICE 14 pin header
- Six general-purpose LEDs, four general-purpose push buttons
- Discrete IDC Expansion ports for all processor peripherals

All sources and tools (compiler, binutils, gnu debugger) needed to create a working uClinux kernel on the Blackfin Processors can be freely obtained from `http://www.blackfin.uclinux.org`. To use the binary rpms, a PC with a Linux distribution like Red Hat or SuSE is needed. Developers who can not install Linux on their PC have a alternative. Cooperative Linux (coLinux) is a relatively new means to provide Linux services on a Windows host. There already exists an out-of-the-box solution that can be downloaded for free from `http://blackfin.uclinux.org/projects/bfin-colinux`. This package comes with a complete Blackfin uClinux distribution, including all user-space applications and a graphical Windows-like installer.

After the installation of the development environment and the decompression of the uClinux distribution, development may start.

[5]Analog Devices, Inc. Blackfin Processors `http://www.analog.com/blackfin`

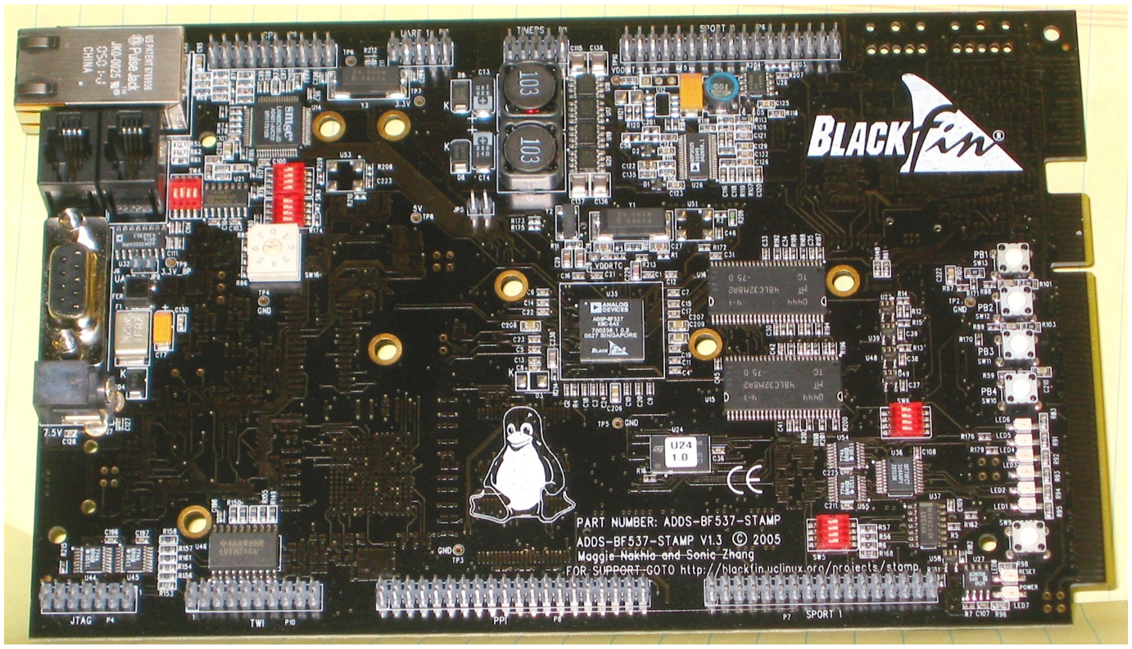

BLACKfin
PART NUMBER: ADDS-BF537-STAMP
ADDS-BF537-STAMP V1.3 © 2005
FOR SUPPORT GOTO http://blackfin.uclinux.org/projects/stamp
JTAG
TWI
PPI
SPORT 1
UART 1
TIMERS
SPORT

Figure 1: BF537-STAMP Board from Analog Devices

6.3 Compiling a kernel & Root Filesystem

First the developer uses the graphical configuration utility to select an appropriate Board Support Package (BSP) for his target hardware. Supported target platforms are STAMP for BF533, BF537, or the EZKIT for the Dual Core Blackfin BF561. Other Blackfin derivatives not listed like BF531, BF532, BF536, or BF534 are also supported but there isn't a default configuration file included.

After the default kernel is configured and successfully compiled, there is a full-featured Linux kernel and a filesystem image that can be downloaded and executed or flashed via NFS, tftp, or Kermit protocol onto the target hardware with the help of preinstalled u-boot boot loader. Once successful, further development can proceed.

6.4 Hello World

A further step could be the creation of a simple Hello World program.

Here is the program `hello.c` as simple as it can be:

```
#include <stdio.h>

int main () {
  printf("Hello World\n");
  return 0;
}
```

The first step is to cross compile `hello.c` on the development host PC:

```
host> bfin-uclinux-gcc -Wl,-elf2flt \
      hello.c -o hello
```

The output executable is `hello`.

When compiling programs that run on the target under the Linux kernel, `bfin-uclinux-gcc` is the compiler used. Executables are linked against the uClibc runtime library. uClibc is a C library for developing embedded Linux systems. It is much smaller than the GNU C Library, but nearly all applications supported by glibc also work perfectly with uClibc. Library function calls like `printf()` invoke a system call, telling the operating system to print a string to stdout, the console. The `elf2flt` command line option tells the linker to generate a flat binary—`elf2flt` converts a fully linked ELF object file created by the toolchain into a binary flat (BFLT) file for use with uClinux.

The next step is to download `hello` to the target hardware. The are many ways to accomplish that. One convenient way is be to place `hello` into a NFS or SAMBA exported file share on the development host, while mounting the share form the target uClinux system. Other alternatives are placing `hello` in a web server's root directory and use the `wget` command on the target board. Or simply use ftp, tftp, or rcp to transfer the executable.

6.5 Debugging in uClinux

Debugging tools in the `hello` case are not a necessity, but as programs become more sophisticated, the availablilty of good debugging tools become a requirement.

Sometimes an application just terminates after being executed, without printing an appropriate error message. Reasons for this are almost infinite, but most of the time it can be traced back to something really simple, e.g. it can not open a file, device driver, etc.

`strace` is a debugging tool which prints out a trace of all the system calls made by a another program. System calls and signals are events that happen at the user/kernel interface. A close examination of this boundary is very useful for bug isolation, sanity checking, and attempting to capture race conditions.

If `strace` does not lead to a quick result, developers can follow the unspectacular way most Linux developers go using `printf` or `printk` to add debug statements in the code and recompile/rerun.

This method can be exhausting. The standard Linux GNU Debugger (GDB) with its graphical front-ends can be used instead to debug user applications. GDB supports single stepping, backtrace, breakpoints, watchpoints, etc. There are several options to have gdb connected to the gdbserver on the target board. Gdb can connect over Ethernet, Serial, or JTAG (rproxy). For debugging in the kernel space, for instance device drivers, developers can use the kgdb Blackfin patch for the gdb debugger.

If a target application does not work because of hidden inefficiencies, profiling is the key to success. OProfile is a system-wide profiler for Linux-based systems, capable of profiling all running code at low overhead. OProfile uses the hardware performance counters of the CPU to enable profiling of a variety of interesting statistics, also including basic time spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

The Blackfin gcc compiler has very favorable performance, a comparison with other gcc compilers can be found here: GCC Code-Size Benchmark Environment (CSiBE). But sometimes it might be necessary to do some hand optimization, to utilize all enhanced instruction capabilities a processor architecture provides. There are a few alternatives: Use Inline assembly, assembly macros, or C-callable assembly.

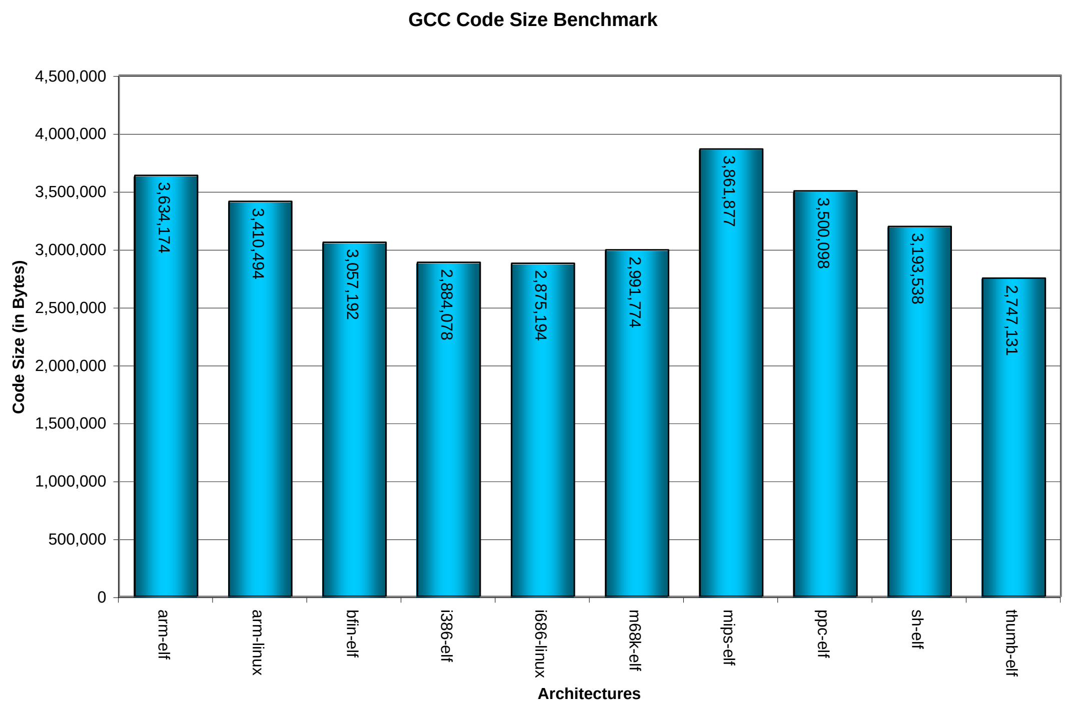


Figure 2: Results from GCC Code-Size Benchmark Environment (CSiBE) Department of Software Engineering, University of Szeged

6.6 C callable assembly

For a C program to be able to call an assembly function, the names of the function must be known to the C program. The function prototype is therefore declared as an external function.

```
extern int minimum(int,int);
```

In the assembly file, the same function name is used as the label at the jump address to which the function call branches. Names defined in C are used with a leading underscore. So the function is defined as follows.

```
.global _minimum;
_minimum:
   R0 = MIN(R0,R1);
   RTS;              /*Return*/
```

The function name must be declared using the `.global` directive in the assembly file to let the assembler and compiler know that its used by a another file. In this case registers R0 and R1 correspond to the first and second function parameter. The function return value is passed in R0. Developers should make themselves comfortable with the C runtime parameter passing model of the used architecture.

6.7 Floating Point & Fractional Floating Point

Since many low cost architectures (like Blackfin) do not include hardware floating point unit (FPU), floating point operations are emulated in software. The gcc compiler supports two variants of soft floating-point support. These variants are implemented in terms of two alternative emulation libraries, selected at compile time.

The two alternative emulation libraries are:

- The default IEEE-754 floating-point library: It is a strictly-conforming variant, which offers less performance, but includes all the input-checking that has been relaxed in the alternative library.
- The alternative fast floating-point library: It is a high-performance variant, which relaxes some of the IEEE rules in the interests of performance. This library assumes that its inputs will be value numbers, rather than Not-a-number values.

The selection of these libraries is controlled with the `-ffast-math` compiler option.

Luckily, most embedded applications do not use floating point.

However, many signal processing algorithms are performed using fractional arithmetic. Unfortunately, C does not have a fixed point fractional data type. However, fractional operations can be implemented in C using integer operations. Most fractional operations must be implemented in multiple steps, and therefore consume many C statements for a single operation, which makes them hard to implement on a general purpose processor. Signal processors directly support single cycle fractional and integer arithmetic, while fractional arithmetic is used for the actual signal processing operations and integer arithmetic is used for control operations such as memory address calculations, loop counters and control variables.

The numeric format in signed fractional notation makes sense to use in all kind of signal processing computations, because it is hard to overflow a fractional result, because multiplying a fraction by a fraction results in a smaller number, which is then either truncated or rounded. The highest full-scale positive fractional number is 0.99999, while the highest full-scale negative number is –1.0. To convert a fractional back to an integer number, the fractional must be multiplied by a scaling factor so the result will be always between $\pm 2^{N-1}$ for signed and 2^N for unsigned integers.

6.8 libraries

The standard uClinux distribution contains a rich set of available C libraries for compression, cryptography, and other purposes. (`openssl`, `libpcap`, `libldap`, `libm`, `libdes`, `libaes`, `zlib`, `libpng`, `libjpeg`, `ncurses`, etc.) The Blackfin/uClinux distribution additionally includes: `libaudio`, `libao`, `libSTL`, `flac`, `tremor`, `libid3tag`, `mpfr`, etc. Furthermore Blackfin/uClinux developers currently incorporate signal processing libraries into uClinux with highly optimized assembly functions to perform all kinds of common signal processing algorithms such as Convolution, FFT, DCT, and IIR/FIR Filters, with low MIPS overhead.

6.9 Application Development

The next step would be the development of the special applications for the target device or the porting of additional software. A lot of development can be done in shell scripts or languages like Perl or Python. Where C programming is mandatory, Linux, with its extraordinary support for protocols and device drivers, provides a powerful environment for the development of new applications.

Example: Interfacing a CMOS Camera Sensor

The Blackfin processor is a very I/O-balanced processor. This means it offers a variety of high-speed serial and parallel peripheral interfaces. These interfaces are ideally designed in a way that they can be operated with very low or no-overhead impact to the processor core, leaving enough time for running the OS and processing the incoming or outgoing data. A Blackfin Processor as an example has multiple, flexible, and independent Direct Memory Access (DMA) controllers. DMA transfers can occur between the processor's internal memory and any of its DMA-capable peripherals. Additionally, DMA transfers can be performed between any of the DMA-capable peripherals and external devices connected to the external memory interfaces, including the SDRAM controller and the asynchronous memory controller.

The Blackfin processor provides, besides other interfaces, a Parallel Peripheral Interface (PPI) that can connect directly to parallel D/A and A/D converters, ITU-R-601/656 video encoders and decoders, and other general-purpose peripherals, such as CMOS camera sensors. The PPI consists of a dedicated input clock pin, up to three frame synchronization pins, and up to 16 data pins.

Figure 3 is an example of how easily a CMOS imaging sensor can be wired to a Blackfin Processor, without the need of additional active hardware components.

Below is example code for a simple program that reads from a CMOS Camera Sensor, assuming a PPI driver is compiled into the kernel or loaded as a kernel module. There are two different PPI drivers available, a generic full-featured driver, supporting various PPI operation modes (`ppi.c`), and a simple PPI Frame Capture Driver (`adsp-ppifcd.c`). The latter is used here. The application opens the PPI device driver, performs some I/O controls (ioctls), setting the number of pixels per line and the number of lines to be captured. After the application invokes the `read` system call, the driver arms the DMA transfer. The start of a new frame is detected by the PPI peripheral, by monitoring the Line- and Frame-Valid strobes. A special correlation between the two signals indicates the start of frame, and kicks off the DMA transfer, capturing pixels-per-line times lines samples. The DMA engine stores the incoming samples at the address allocated by the application. After the transfer is finished, execution returns to the application. The image is then converted into the PNG (Portable Network Graphic) format, utilizing `libpng` included in the uClinux distribution. The converted image is then written to `stdout`. Assuming the compiled program executable is called `readimg`, a command line to execute the program, writing the converted output image to a file, can look like following:

```
root:~> readimg > /var/image.png
```

Example: Reading from a CMOS Camera Sensor

Audio, Video, and Still-image silicon products widely use an I2C-compatible Two Wire Interface (TWI) as a system configuration bus. The configuration bus allows a system master to gain access over device internal configuration registers such as brightness. Usually, I2C devices are controlled by a kernel driver. But it is also possible to access all devices on an adapter from user space, through the `/dev` interface. The following example shows how to write a value of 0x248 into register 9 of a I2C slave device identified by `I2C_DEVID`:

```
#define I2C_DEVID (0xB8>>1)
#define I2C_DEVICE "/dev/i2c-0"


i2c_write_register(I2C_DEVICE,
I2C_DEVID,9,0x0248);
```

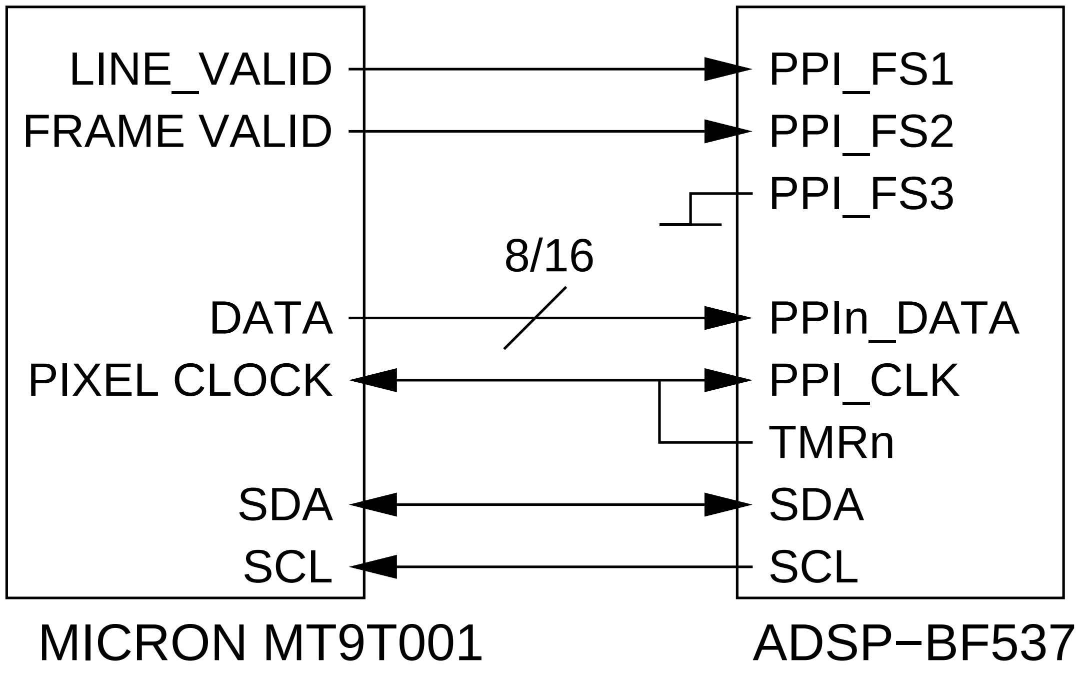


Figure 3: Micron CMOS Imager gluelessly connected to Blackfin

Example: Writing configuration data to e.g. a CMOS Camera Sensor

The power of Linux is the inexhaustible number of applications released under various open source licenses that can be cross compiled to run on the embedded uClinux system. Cross compiling can be sometimes a little bit tricky, that's why it is discussed here.

6.10 Cross compiling

Linux or UNIX is not a single platform, there is a wide range of choices. Most programs distributed as source code come with a so-called *configure* script. This is a shell script that must be run to recognize the current system configuration, so that the correct compiler switches, library paths, and tools will be used. When there isn't a configure script, the developer can manually modify the Makefile to add target-processor-specific changes, or can integrate it into the uClinux distribution. Detailed instructions can be found here. The configure script is usually a big script, and it takes quite a while to execute. When this script is created from recent autoconf releases, it will work for Blackfin/uClinux with minor or no modifications.

The configure shell script inside a source package can be executed for cross compilation using following command line:

```
host> CC='bfin-uclinux-gcc -O2 \
   -Wl,-elf2flt' ./configure   \
   --host=bfin-uclinux         \
   --build=i686-linux
```

Alternatively:

```
host> ./configure        \
  --host=bfin-uclinux    \
  --build=i686-linux     \
  LDFLAGS='-Wl,-elf2flt' \
  CFLAGS=-O2
```

```
#define WIDTH           1280
#define HEIGHT          1024

int main( int argc, char *argv[] ) {
    int fd;
    char * buffer;

    /* Allocate memory for the raw image */
    buffer = (char*) malloc (WIDTH * HEIGHT);

    /* Open /dev/ppi */
    fd = open("/dev/ppi0", O_RDONLY,0);
    if (fd == -1) {
        printf("Could not open dev\/ppi\n");
        free(buffer);
        exit(1);
    }

    ioctl(fd, CMD_PPI_SET_PIXELS_PER_LINE, WIDTH);
    ioctl(fd, CMD_PPI_SET_LINES_PER_FRAME, HEIGHT);

    /* Read the raw image data from the PPI */
    read(fd, buffer, WIDTH * HEIGHT);

    put_image_png (buffer, WIDTH, HEIGHT)

    close(fd);   /* Close PPI */
}

/*
 * convert image to png and write to stdout
 */
void put_image_png (char *image, int width, int height) {
    int y;
    char *p;
    png_infop info_ptr;

    png_structp png_ptr = png_create_write_struct (PNG_LIBPNG_VER_STRING,
                NULL, NULL, NULL);

    info_ptr = png_create_info_struct (png_ptr);

    png_init_io (png_ptr, stdout);

    png_set_IHDR (png_ptr, info_ptr, width, height,
                  8, PNG_COLOR_TYPE_GRAY, PNG_INTERLACE_NONE,
                  PNG_COMPRESSION_TYPE_DEFAULT, PNG_FILTER_TYPE_DEFAULT);

    png_write_info (png_ptr, info_ptr);
    p = image;

    for (y = 0; y < height; y++) {
        png_write_row (png_ptr, p);
        p += width;
    }
    png_write_end (png_ptr, info_ptr);
    png_destroy_write_struct (&png_ptr, &info_ptr);
}
```

Figure 4: read.c file listing

```
#define I2C_SLAVE_ADDR 0x38 /* Randomly picked */

int i2c_write_register(char * device, unsigned char client, unsigned char reg,
                       unsigned short value) {
    int    addr = I2C_SLAVE_ADDR;
    char   msg_data[32];
    struct i2c_msg msg = { addr, 0, 0, msg_data };
    struct i2c_rdwr_ioctl_data rdwr = { &msg, 1 };

    int fd,i;

    if ((fd = open(device, O_RDWR)) < 0) {
        fprintf(stderr, "Error: could not open %s\n", device);
        exit(1);
    }

    if (ioctl(fd, I2C_SLAVE, addr) < 0) {
        fprintf(stderr, "Error: could not bind address %x \n", addr);
    }

    msg.len   = 3;
    msg.flags = 0;
    msg_data[0] = reg;
    msg_data[2] = (0xFF & value);
    msg_data[1] = (value >> 8);
    msg.addr = client;

    if (ioctl(fd, I2C_RDWR, &rdwr) < 0) {
        fprintf(stderr, "Error: could not write \n");
    }

    close(fd);
    return 0;
}
```

Figure 5: Application to write configuration data to a CMOS Sensor

There are at least two causes able to stop the running script: some of the files used by the script are too old, or there are missing tools or libraries. If the supplied scripts are too old to execute properly for `bfin-uclinux`, or they don't recognize `bfin-uclinux` as a possible target, the developer will need to replace `config.sub` with a more recent version from e.g. an up-to-date gcc source directory. Only in very few cases cross compiling is not supported by the `configure.in` script manually written by the author and used by autoconf. In this case latter file can be modified to remove or change the failing test case.

7 Network Oscilloscope

The Network Oscilloscope Demo is one of the sample applications, besides the VoIP Linphone Application or the Networked Audio Player, included in the Blackfin/uClinux distribution. The purpose of the Network Oscilloscope Project is to demonstrate a simple remote GUI (Graphical User Interface) mechanism to share access and data distributed over a TCP/IP network. Furthermore, it demonstrates the integration of several open source projects and libraries as building blocks into single application. For instance `gnuplot`, a portable command-line driven interactive data file and function plotting utility, is used to generate graphical data plots, while `thttpd`, a CGI (Common Gateway Interface) capable web server, is servicing incoming HTTP requests. CGI is typically used to generate dynamic webpages. It's a simple protocol to communicate between web forms and a specified program. A CGI script can be written in any language, including C/C++, that can read `stdin`, write to `stdout`, and read environment variables.

The Network Oscilloscope works as following. A remote web browser contacts the HTTP server running on uClinux where the CGI script resides, and asks it to run the program. Parameters from the HTML form such as sample frequency, trigger settings, and display options are passed to the program through the environment. The called program samples data from a externally connected Analog-to-Digital Converter (ADC) using a Linux device driver (`adsp-spiadc.c`). Incoming samples are preprocessed and stored in a file. The CGI program then starts gnuplot as a process and requests generation of a PNG or JPEG image based on the sampled data and form settings. The webserver takes the output of the CGI program and tunnels it through to the web browser. The web browser displays the output as an HTML page, including the generated image plot.

A simple C code routine can be used to supply data in response to a CGI request.

Example: Simple CGI Hello World application

8 Real-time capabilities of uClinux

Since Linux was originally developed for server and desktop usage, it has no hard real-time capabilities like most other operating systems of comparable complexity and size. Nevertheless, Linux and in particular, uClinux has excellent so-called *soft* real-time capabilities. This means that while Linux or uClinux cannot guarantee certain interrupt or scheduler latency compared with other operating systems of similar complexity, they show very favorable performance characteristics. If one needs a so-called hard real-time system that can guarantee scheduler or interrupt latency time, there are a few ways to achieve such a goal:

Provide the real-time capabilities in the form of an underlying minimal real-time kernel such as RT-Linux (`http://www.rtlinux.org`) or RTAI (`http://www.rtai.org`). Both solutions use a small real-time kernel that runs Linux as a real-time task with lower priority. Programs that need predictable real time are designed to run on the real-time kernel and are specially coded to do so. All other tasks and services run on top of the Linux kernel and can utilize everything that Linux can provide. This approach can guarantee deterministic interrupt latency while preserving the flexibility that Linux provides.

For the initial Blackfin port, included in Xenomai v2.1, the worst-case scheduling latency observed so far with userspace Xenomai threads on a Blackfin BF533 is slightly lower than 50 us under load, with an expected margin of improvement of 10–20 us, in the future.

Xenomai and RTAI use Adeos as a underlying Hardware Abstraction Layer (HAL). Adeos is a real time enabler for the Linux kernel. To this end, it enables multiple prioritized O/S domains to exist simultaneously on the same hardware, connected through an interrupt pipeline.

Xenomai as well as Adeos has been ported to the Blackfin architecture by Philippe Gerum who leads both projects. This development has been significantly sponsored by Openwide, a specialist in embedded and real time solutions for Linux.

Nevertheless in most cases, hard real-time is not needed, particularly for consumer multimedia applications, in which the time constraints are dictated by the abilities of the user to recognize glitches in audio and video. Those physically detectable constraints that have to be met normally lie in the area of milliseconds, which is no big problem on fast chips like the Blackfin Processor. In Linux kernel 2.6.x, the new stable kernel release, those qualities have even been improved with the introduction of the new O(1) scheduler.

Figures below show the context switch time for a default Linux 2.6.x kernel running on Blackfin/uClinux:

Context Switch time was measured with `lat_ctx` from lmbench. The processes are connected in a ring of Unix pipes. Each process reads a token from its pipe, possibly does some work, and then writes the token to the next process. As number of processes increases, effect of cache is less. For 10 processes the average context switch time is 16.2us with a standard deviation of .58, 95% of time is under 17us.

9 Conclusion

Blackfin Processors offer a good price/performance ratio (800 MMAC @ 400 MHz for less than (US)$5/unit in quantities), advanced power management functions, and small mini-BGA packages. This represents a very low-power, cost- and space-efficient solution. The Blackfin's advanced DSP and multimedia capabilities qualify it not only for audio and video appliances, but also for all kinds of industrial, automotive, and communication devices. Development tools are well tested and documented, and include everything necessary to get started and successfully finished in time. Another advantage of the Blackfin Processor in combination with uClinux is the availability of a wide range of applications, drivers, libraries and protocols, often as open source or free software. In most cases, there is only basic cross compilation necessary to get that software up and running. Combine this with such invaluable tools as Perl, Python, MySQL, and PHP, and developers have the opportunity to develop even the most demanding feature-rich applications in a very short time frame, often with enough processing power left for future improvements and new features.

10 Legal

This work represents the view of the authors and does not necessarily represent the view of Analog Devices, Inc.

Linux is registered trademark of Linus Torvalds. uClinux is trademark of Arcturus Networks Inc. SGI is trademark of Silicon Graphics, Inc. ARM is a registered trademark of ARM Limited. Blackfin is a registered trademark of Analog Devices Inc. IBM is a registered trademark of International Business Machines Corporation. UNIX is a registered trademark of The Open Group. Red Hat is registered trademark of Red Hat, Inc. SuSE is registered trademark of Novell Inc.

All other trademarks belong to their respective owners.

Hrtimers and Beyond: Transforming the Linux Time Subsystems

Thomas Gleixner
linutronix
`tglx@linutronix.de`

Douglas Niehaus
University of Kansas
`niehaus@eecs.ku.edu`

Abstract

Several projects have tried to rework Linux time and timers code to add functions such as high-precision timers and dynamic ticks. Previous efforts have not been generally accepted, in part, because they considered only a subset of the related problems requiring an integrated solution. These efforts also suffered significant architecture dependence creating complexity and maintenance costs. This paper presents a design which we believe provides a generally acceptable solution to the complete set of problems with minimum architecture dependence.

The three major components of the design are hrtimers, John Stulz's new timeofday approach, and a new component called clock events. Clock events manages and distributes clock events and coordinates the use of clock event handling functions. The hrtimers subsystem has been merged into Linux 2.6.16. Although the name implies "high resolution" there is no change to the tick based timer resolution at this stage. John Stultz's timeofday rework addresses jiffy and architecture independent time keeping and has been identified as a fundamental preliminary for high resolution timers and tickless/dynamic tick solutions. This paper provides details on the hrtimers implementation and describes how the clock events component will complement and complete the hrtimers and timeofday components to create a solid foundation for architecture independent support of high-resolution timers and dynamic ticks.

1 Introduction

Time keeping and use of clocks is a fundamental aspect of operating system implementation, and thus of Linux. Clock related services in operating systems fall into a number of different categories:

- time keeping
- clock synchronization
- time-of-day representation
- next event interrupt scheduling
- process and in-kernel timers
- process accounting
- process profiling

These service categories exhibit strong interactions among their semantics at the design level and tight coupling among their components at the implementation level.

Hardware devices capable of providing clock sources vary widely in their capabilities, accuracy, and suitability for use in providing the desired clock services. The ability to use a given hardware device to provide a particular clock service also varies with its context in a uniprocessor or multi-processor system.

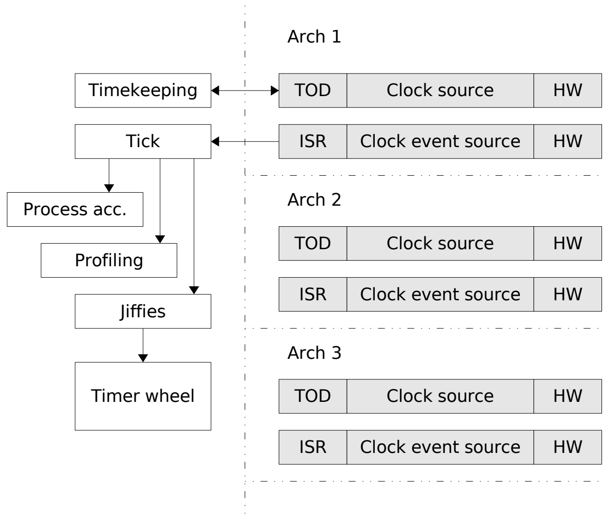


Figure 1: Linux Time System

Figure 1 shows the current software architecture of the clock related services in a vanilla 2.6 Linux system. The current implementation of clock related services in Linux is strongly associated with individual hardware devices, which results in parallel implementations for each architecture containing considerable amounts of essentially similar code. This code duplication across a large number of architectures makes it difficult to change the semantics of the clock related services or to add new features such as high resolution timers or dynamic ticks because even a simple change must be made in so many places and adjusted for so many implementations. Two major factors make implementing changes to Linux clock related services difficult: (1) the lack of a generic abstraction layer for clock services and (2) the assumption that time is tracked using periodic timer ticks (jiffies) that is strongly integrated into much of the clock and timer related code.

2 Previous Efforts

A number of efforts over many years have addressed various clock related services and functions in Linux including various approaches to high resolution time keeping and event scheduling. However, all of these efforts have encountered significant difficulty in gaining broad acceptance because of the breadth of their impact on the rest of the kernel, and because they generally addressed only a subset of the clock related services in Linux.

Interestingly, all those efforts have a common design pattern, namely the attempt to integrate new features and services into the existing clock and timer infrastructure without changing the overall design.

There are no projects to our knowledge which attempt to solve the *complete* problem as we understand and have described it. All existing efforts, in our view, address only a part of the whole problem as we see it, which is why, in our opinion, the solutions to their target problems are more complex than under our proposed architecure, and are thus less likely to be accepted into the main line kernel.

3 Required Abstractions

The attempt to integrate high resolution timers into Ingo Molnar's real-time preemption patch led to a thorough analysis of the Linux timer and clock services infrastructure. While the comprehensive solution for addressing the overall problem is a large-scale task it can be separated into different problem areas.

- clock sources management for time keeping

- clock synchronization
- time-of-day representation
- clock event management for scheduling next event interrupts
- eliminating the assumption that timers are supported by periodic interrupts and expressed in units of jiffies

These areas of concern are largely independent and can thus be addressed more or less independently during implementation. However, the important points of interaction among them must be considered and supported in the overall design.

3.1 Clock Source Management

An abstraction layer and associated API are required to establish a common code framework for managing various clock sources. Within this framework, each clock source is required to maintain a representation of time as a monotonically increasing value. At this time, nanoseconds are the favorite choice for the time value units of a clock source. This abstraction layer allows the user to select among a range of available hardware devices supporting clock functions when configuring the system and provides necessary infrastructure. This infrastructure includes, for example, mathematical helper functions to convert time values specific to each clock source, which depend on properties of each hardware device, into the common human-oriented time units used by the framework, i.e. nanoseconds. The centralization of this functionality allows the system to share significantly more code across architectures. This abstraction is already addressed by John Stultz's work on a Generic Time-of-day subsystem [5].

3.2 Clock Synchronization

Crystal driven clock sources tend to be imprecise due to variation in component tolerances and environmental factors, such as temperature, resulting in slightly different clock tick rates and thus, over time, different clock values in different computers. The Network Time Protocol (NTP) and more recently GPS/GSM based synchronization mechanisms allow the correction of system time values and of clock source drift with respect to a selected standard. Value correction is applied to the monotonically increasing value of the hardware clock source. This is an optional functionality as it can only be applied when a suitable reference time source is available. Support for clock synchronization is a separate component from those discussed here. There is work in progress to rework the current mechanism by John Stultz and Roman Zippel.

3.3 Time-of-day Representation

The monotonically increasing time value provided by many hardware clock sources cannot be set. The generic interface for time-of-day representation must thus compensate for drift as an offset to the clock source value, and represent the time-of-day (calendar or wall clock time) as a function of the clock source value. The drift offset and parameters to the function converting the clock source value to a wall clock value can set by manual interaction or under control of software for synchronization with external time sources (e.g. NTP).

It is important to note that the current Linux implementation of the time keeping component is the reverse of the proposed solution. The internal time representation tracks the time-of-day time fairly directly and derives the monotonically increasing nanosecond time value from it.

This is a relic of software development history and the GTOD/NTP work is already addressing this issue.

3.4 Clock Event Management

While clock sources provide read access to the monotonically increasing time value, clock event sources are used to schedule the next event interrupt(s). The next event is currently defined to be periodic, with its period defined at compile time. The setup and selection of the event sources for various event driven functionalities is hardwired into the architecture dependent code. This results in duplicated code across all architectures and makes it extremely difficult to change the configuration of the system to use event interrupt sources other than those already built into the architecture. Another implication of the current design is that it is necessary to touch all the architecture-specific implementations in order to provide new functionality like high resolution timers or dynamic ticks.

The clock events subsystem tries to address this problem by providing a generic solution to manage clock event sources and their usage for the various clock event driven kernel functionalities. The goal of the clock event subsystem is to minimize the clock event related architecture dependent code to the pure hardware related handling and to allow easy addition and utilization of new clock event sources. It also minimizes the duplicated code across the architectures as it provides generic functionality down to the interrupt service handler, which is almost inherently hardware dependent.

3.5 Removing Tick Dependencies

The strong dependency of Linux timers on using the the periodic tick as the time source and representation was one of the main problems faced when implementing high resolution timers and variable interval event scheduling. All attempts to reuse the cascading timer wheel turned out to be incomplete and inefficient for various reasons. This led to the implementation of the *hrtimers* (former ktimers) subsystem. It provides the base for precise timer scheduling and is designed to be easily extensible for high resolution timers.

4 hrtimers

The current approach to timer management in Linux does a good job of satisfying an extremely wide range of requirements, but it cannot provide the quality of service required in some cases precisely because it must satisfy such a wide range of requirements. This is why the essential first step in the approach described here is to implement a new timer subsystem to complement the existing one by assuming a subset of its existing responsibilities.

4.1 Why a New Timer Subsystem?

The Cascading Timer Wheel (CTW), which was implemented in 1997, replaced the original time ordered double linked list to resolve the scalability problem of the linked list's O(N) insertion time. It is based on the assumption that the timers are supported by a periodic interrupt (jiffies) and that the expiration time is also represented in jiffies. The difference in time value (delta) between now (the current system time) and the timer's expiration value is used as an index into the CTW's logarithmic array of arrays. Each array within the CTW represents the set of timers placed within a region of the system time line, where the size of the array's regions grow exponentially. Thus, the further into the

array	start	end	granularity
1	1	256	1
2	257	16384	256
3	16385	1048576	16384
4	1048577	67108864	1048576
5	67108865	4294967295	67108864

Table 1: Cascading Timer Wheel Array Ranges

future a timer's expiration value lies, the larger the region of the time line represented by the array in which it is stored. The CTW groups timers into 5 categories. Note that each CTW array represents a range of jiffy values and that more than one timer can be associated with a given jiffy value.

Table 1 shows the properties of the different timer categories. The first CTW category consists of n1 entries, where each entry represents a single jiffy. The second category consists of n2 entries, where each entry represents n1*n2 jiffies. The third category consists of n3 entries, where each entry represents n1*n2*n3 jiffies. And so forth. The current kernel uses n1=256 and n2..n5 = 64. This keeps the number of hash table entries in a reasonable range and covers the future time line range from 1 to 4294967295 jiffies.

The capacity of each category depends on the size of a jiffy, and thus on the periodic interrupt interval. While the 10 ms tick period in 2.4 kernels implied 2560ms for the CTW first category, this was reduced to 256ms in the early 2.6 kernels (1 ms tick) and readjusted to 1024ms when the HZ value was set to 250. Each CTW category maintains an time index counter which is incremented by the "wrapping" of the lower category index which occurs when its counter increases to the point where its range overlaps that of the higher category. This triggers a "cascade" where timers from the matching entry in the higher category have to be removed and reinserted into the lower category's finer-grained entries. Note that in the first CTW category the timers are time-sorted with jiffy resolution.

While the CTW's O(1) insertion and removal is very efficient, timers with an expiration time larger than the capacity of the first category have to be cascaded into a lower category at least once. A single step of cascading moves many timers and it has to be done with interrupts disabled. The cascading operation can thus significantly increase maximum latencies since it occasionally moves very large sets of timers. The CTW thus has excellent average performance but unacceptable worst case performance. Unfortunately the worst case performance determines its suitability for supporting high resolution timers.

However, it is important to note that the CTW is an excellent solution (1) for timers having an expiration time lower than the capacity of the primary category and (2) for timers which are removed before they expire or have to be cascaded. This is a common scenario for many long-term protocol-timeout related timers which are created by the networking and I/O subsystems.

The KURT-Linux project at the University of Kansas was the first to address implementing high resolution timers in Linux [4]. Its concentration was on investigating various issues related to using Linux for real-time computing. The UTIME component of KURT-Linux experimented with a number of data structures to support high resolution timers, including both separate implementations and those integrated with general purpose timers. The HRT project began as a fork of UTIME code [1]. both projects added a sub-jiffy time tracking component to increase resolution, and when integrating support with the CTW, sorted timers within a given jiffy on the basis of the subjiffy value.

This increased overhead involved with cascading due to the O(N) sorting time. The experience of both projects demonstrated that timer management overhead was a significant factor, and that the necessary changes in the timer code were quite scattered and intrusive. In summary, the experience of both projects demonstrated that separating support for high resolution and longer-term generic (CTW) timers was necessary and that a comprehensive restructuring of the timer-related code would be required to make future improvements and additions to timer-related functions possible. The *hrtimers* design and other aspects of the architecture described in this paper was strongly motivated by the lessons derived from both previous projects.

4.2 Solution

As a first step we categorized the timers into two categories:

Timeouts: Timeouts are used primarily by networking and device drivers to detect when an event (I/O completion, for example) does not occur as expected. They have low resolution requirements, and they are almost always removed before they actually expire.

Timers: Timers are used to schedule ongoing events. They can have high resolution requirements, and usually expire. Timers are mostly related to applications (user space interfaces)

The timeout related timers are kept in the existing timer wheel and a new subsystem optimized for (high resolution) timer requirements *hrtimers* was implemented.

hrtimers are entirely based on human time units: nanoseconds. They are kept in a time sorted, per-CPU list, implemented as a red-black tree. Red-black trees provide O(log(N)) insertion and removal and are considered to be efficient enough as they are already used in other performance critical parts of the kernel e.g. memory management. The timers are kept in relation to time bases, currently `CLOCK_MONOTONIC` and `CLOCK_REALTIME`, ordered by the absolute expiration time. This separation allowed to remove large chunks of code from the POSIX timer implementation, which was necessary to recalculate the expiration time when the clock was set either by settimeofday or NTP adjustments.

hrtimers went through a couple of revision cycles and were finally merged into Linux 2.6.16. The timer queues run from the normal timer softirq so the resulting resolution is not better than the previous timer API. All of the structure is there to do better once the other parts of the overall timer code rework are in place.

After adding *hrtimers* the Linux time(r) system looks like this:

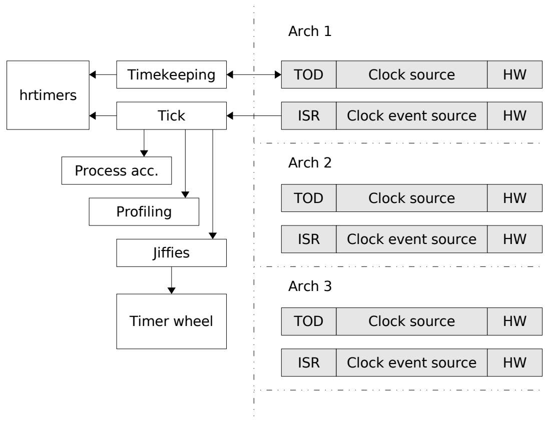


Figure 2: Linux time system + htimers

4.3 Further Work

The primary purpose of the separate implementation for the high resolution timers, discussed in Section 7, is to improve their support by eliminating the overhead and variable latency associated with the CTW. However, it is also important to note that this separation also creates an opportunity to improve the CTW behavior in supporting the remaining timers. For example, using a coarser CTW granularity may lower overhead by reducing the number of timers which are cascaded, given that an even larger percentage of CTW timers would be canceled under an architecture supporting high resolution timers separately. However, while this is an interesting possibility, it is currently a speculation that must be tested.

5 Generic Time-of-day

The Generic Time-of-day subsystem (GTOD) is a project led by John Stultz and was presented at OLS 2005. Detailed information is available from the OLS 2005 proceedings [5]. It contains the following components:

- Clock source management
- Clock synchronization
- Time-of-day representation

GTOD moves a large portion of code out of the architecture-specific areas into a generic management framework, as illustrated in Figure 3. The remaining architecture-dependent code is mostly limited to the direct hardware interface and setup procedures. It allows simple sharing of clock sources across architectures and allows the utilization of non-standard clock source hardware. GTOD is work in progress and intends to produce set of changes which can be adopted step by step into the main-line kernel.

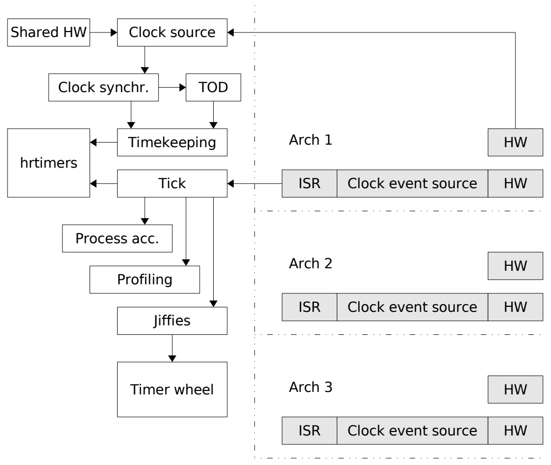


Figure 3: Linux time system + htimers + GTOD

6 Clock Event Source Abstraction

Just as it was necessary to provide a general abstraction for clock sources in order to move a significant amount of code into the architecture independent area, a general framework for managing clock event sources is also required in the architecture independent section of the source under the architecture described here. Clock event sources provide either periodic or individual programmable events. The management layer provides the infrastructure for registering event sources and manages the distribution of the events for the various clock related services. Again, this reduces the amount of essentially duplicate code across the architectures and allows cross-architecture sharing of hardware support code and the easy addition of non-standard clock sources.

The management layer provides interfaces for *hrtimers* to implement high resolution timers and also builds the base for a generic dynamic tick implementation. The management layer supports these more advanced functions only when appropriate clock event sources have been registered, otherwise the traditional periodic tick based behavior is retained.

6.1 Clock Event Source Registration

Clock event sources are registered either by the architecture dependent boot code or at module insertion time. Each clock event source fills a data structure with clock-specific property parameters and callback functions. The clock event management decides, by using the specified property parameters, the set of system functions a clock event source will be used to support. This includes the distinction of per-CPU and per-system global event sources.

System-level global event sources are used for the Linux periodic tick. Per-CPU event source are used to provide local CPU functionality such as process accounting, profiling, and high resolution timers. The `clock_event` data structure contains the following elements:

- `name`: clear text identifier
- `capabilities`: a bit-field which describes the capabilities of the clock event source and hints about the preferred usage
- `max_delta_ns`: the maximum event delta (offset into future) which can be scheduled
- `min_delta_ns`: the minimum event delta which can be scheduled
- `mult`: multiplier for scaled math conversion from nanoseconds to clock event source units
- `shift`: shift factor for scaled math conversion from nanoseconds to clock event source units
- `set_next_event`: function to schedule the next event
- `set_mode`: function to toggle the clock event source operating mode (periodic / one shot)
- `suspend`: function which has to be called before suspend
- `resume`: function which has to be called before resume
- `event_handler`: function pointer which is filled in by the clock event management code. This function is called from the event source interrupt
- `start_event`: function called before the `event_handler` function in case that the clock event layer provides the interrupt handler
- `end_event`: function called after the `event_handler` function in case that the clock event layer provides the interrupt handler
- `irq`: interrupt number in case the clock event layer requests the interrupt and provides the interrupt handler
- `priv`: pointer to clock source private data structures

The clock event source can delegate the interrupt setup completely to the management layer. It depends on the type of interrupt which is associated with the event source. This is possible for the PIT on the i386 architecture, for example, because the interrupt in question is handled by the generic interrupt code and can be initialized via setup_irq. This allows us to completely remove the timer interrupt handler from the i386 architecture-specific area and move the modest amount of hardware-specific code into appropriate source files. The hardware-specific routines are called before and after the event handling code has been executed.

In case of the Local APIC on i386 and the Decrementer on PPC architectures, the interrupt handler must remain in the architecture-specific code as it can not be setup through the standard interrupt handling functions. The clock management layer provides the function which has to be called from the hardware level handler in a function pointer in the clock source description structure. Even in this case the shared code of the timer interrupt is removed from the architecture-specific implementation and the event distribution is managed by the generic clock event code. The Clock Events subsystem also has support code for clock event sources which do not provide a periodic mode; e.g. the Decrementer on PPC or match register based event sources found in various ARM SoCs.

6.2 Clock Event Distribution

The clock event layer provides a set of predefined functions, which allow the association of various clock event related services to a clock event source.

The current implementation distributes events for the following services:

- periodic tick
- process accounting
- profiling
- next event interrupt (*e.g.* high resolution timers, dynamic tick)

6.3 Interfaces

The clock event layer API is rather small. Aside from the clock event source registration interface it provides functions to schedule the next event interrupt, clock event source notification service, and support for suspend and resume.

6.4 Existing Implementations

At the time of this writing the base framework code and the conversion of i386 to the clock event layer is available and functional.

The clock event layer has been successfully ported to ARM and PPC, but support has not been continued due to lack of human resources.

6.5 Code Size Impact

The framework adds about 700 lines of code which results in a 2KB increase of the kernel binary size.

The conversion of i386 removes about 100 lines of code. The binary size decrease is in the range of 400 bytes.

We believe that the increase of flexibility and the avoidance of duplicated code across architectures justifies the slight increase of the binary size.

The first goal of the clock event implementation was to prove the feasibility of the approach. There is certainly room for optimizing the size impact of the framework code, but this is an issue for further development.

6.6 Further Development

The following work items are planned:

- Streamlining of the code
- Revalidation of the clock distribution decisions
- Support for more architectures
- Dynamic tick support

6.7 State of Transformation

The clock event layer adds another level of abstraction to the Linux subsystem related to time keeping and time-related activities, as illustrated in Figure 4. The benefit of adding the abstraction layer is the substantial reduction in architecture-specific code, which can be seen most clearly by comparing Figures 3 and 4.

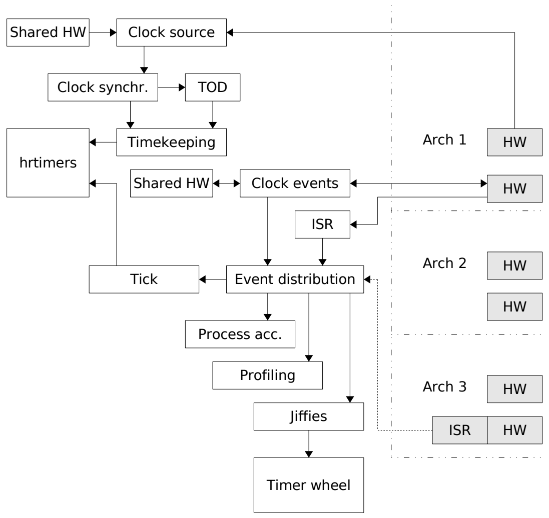


Figure 4: Linux time system + htimers + GTOD + clock events

7 High Resolution Timers

The inclusion of the clock source and clock event source management and abstraction layers provides now the base for high resolution support for *hrtimers*.

While previous attempts of high resolution timer implementations needed modification all over the kernel source tree, the *hrtimers* based implementation only changes the *hrtimers* code itself. The required change to enable high resolution timers for an architecture which is supported by the Generic Time-of-day and the clock event framework is the inclusion of a single line in the architecture specific Kconfig file.

The next event modifications remove the implicit but strong binding of hrtimers to jiffy tick boundaries. When the high resolution extension is disabled the clock event distribution code works in the original periodic mode and hrtimers are bound to jiffy tick boundaries again.

8 Implementation

While the base functionality of *hrtimers* remains unchanged, additional functionality had to be added.

- Management function to switch to high resolution mode late in the boot process.
- Next event scheduling
- Next event interrupt handler
- Separation of the *hrtimers* queue from the timer wheel softirq

During system boot it is not possible to use the high resolution timer functionality, while making it possible would be difficult and would serve no useful function. The initialization of the clock event framework, the clock source framework and *hrtimers* itself has to be done and appropriate clock sources and clock event sources have to be registered before the high resolution functionality can work. Up to the point where hrtimers are initialized, the system works in the usual low resolution periodic mode. The clock source and the clock event source layers provide notification functions which inform *hrtimers* about availability of new hardware. *hrtimers* validates the usability of the registered clock sources and clock event sources before switching to high resolution mode. This ensures also that a kernel which is configured for high resolution timers can run on a system which lacks the necessary hardware support.

The time ordered insertion of *hrtimers* provides all the infrastructure to decide whether the event source has to be reprogrammed when a timer is added. The decision is made per timer base and synchronized across timer bases in a support function. The design allows the system to utilize separate per-CPU clock event sources for the per-CPU timer bases, but mostly only one reprogrammable clock event source per-CPU is available. The high resolution timer does not support SMP machines which have only global clock event sources.

The next event interrupt handler is called from the clock event distribution code and moves expired timers from the red-black tree to a separate double linked list and invokes the softirq handler. An additional mode field in the *hrtimer* structure allows the system to execute callback functions directly from the next event interrupt handler. This is restricted to code which can safely be executed in the hard interrupt context and does not add the timer back to the red-black tree. This applies, for example, to the common case of a wakeup function as used by nanosleep. The advantage of executing the handler in the interrupt context is the avoidance of up to two context switches—from the interrupted context to the softirq and to the task which is woken up by the expired timer. The next event interrupt handler also provides functionality which notifies the clock event distribution code that a requested periodic interval has elapsed. This allows to use a single clock event source to schedule high resolution timer and periodic events e.g. jiffies tick, profiling, process accounting. This has been proved to work with the PIT on i386 and the Incrementer on PPC.

The softirq for running the *hrtimer* queues and executing the callbacks has been separated from the tick bound timer softirq to allow accurate delivery of high resolution timer signals which are used by itimer and POSIX interval timers. The execution of this softirq can still be delayed by other softirqs, but the overall latencies have been significantly improved by this separation.

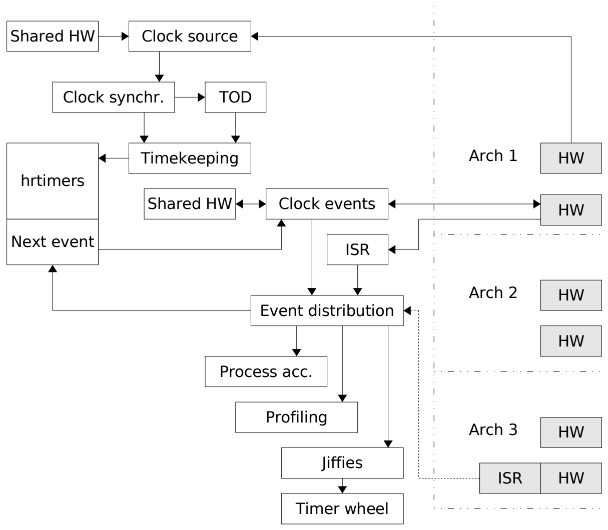


Figure 5: Linux time system + htimers + GTOD + clock events + high resolution timers

8.1 Accuracy

All tests have been run on a Pentium III 400MHz based PC. The tables show comparisons of vanilla Linux 2.6.16, Linux-2.6.16-hrt5 and Linux-2.6.16-rt12. The tests for intervals less than the jiffy resolution have not been run on vanilla Linux 2.6.16. The test thread runs in all cases with SCHED_FIFO and priority 80.

Test case: `clock_nanosleep(TIME_ ABSTIME)`, Interval 10000 microseconds, 10000 loops, no load.

Kernel	min	max	avg
2.6.16	24	4043	1989
2.6.16-hrt5	12	94	20
2.6.16-rt12	6	40	10

Test case: `clock_nanosleep(TIME_ABSTIME)`, Interval 10000 micro seconds, 10000 loops, 100% load.

Kernel	min	max	avg
2.6.16	55	4280	2198
2.6.16-hrt5	11	458	55
2.6.16-rt12	16		

Test case: POSIX interval timer, Interval 10000 micro seconds, 10000 loops, no load.

Kernel	min	max	avg
2.6.16	21	4073	2098
2.6.16-hrt5	22	120	35
2.6.16-rt12	20	60	31

Test case: POSIX interval timer, Interval 10000 micro seconds, 10000 loops, 100% load.

Kernel	min	max	avg
2.6.16	82	4271	2089
2.6.16-hrt5	31	458	53
2.6.16-rt12	21	70	35

Test case: `clock_nanosleep(TIME_ABSTIME)`, Interval 500 micro seconds, 100000 loops, no load.

Kernel	min	max	avg
2.6.16-hrt5	5	108	24
2.6.16-rt12	5	48	7

Test case: `clock_nanosleep(TIME_ABSTIME)`, Interval 500 micro seconds, 100000 loops, 100% load.

Kernel	min	max	avg
2.6.16-hrt5	9	684	56
2.6.16-rt12	10	60	22

Test case: POSIX interval timer, Interval 500 micro seconds, 100000 loops, no load.

Kernel	min	max	avg
2.6.16-hrt5	8	119	22
2.6.16-rt12	12	78	16

Test case: POSIX interval timer, Interval 500 micro seconds, 100000 loops, 100% load.

Kernel	min	max	avg
2.6.16-hrt5	16	489	58
2.6.16-rt12	12	95	29

The real-time preemption kernel results are significantly better under high load due to the general low latencies for high priority real-time tasks. Aside from the general latency optimizations, further improvements were implemented specifically to optimize the high resolution timer behavior.

Separate threads for each softirq. Long lasting softirq callback functions e.g. in the networking code do not delay the delivery of *hrtimer* softirqs.

Dynamic priority adjustment for high resolution timer softirqs. Timers store the priority of the task which inserts the timer and the next event interrupt code raises the priority of the *hrtimer* softirq when a callback function for a high priority thread has to be executed. The softirq lowers its priority automatically after the execution of the callback function.

9 Dynamic Ticks

We have not yet done a dynamic tick implementation on top of the existing framework, but we considered the requirements for such an implementation in every design step.

The framework does not solve the general problem of dynamic ticks: how to find the next expiring timer in the timer wheel. In the worst case the code has to walk through a large number of hash buckets. This can not be changed without changing the basic semantics and implementation details of the timer wheel code.

The next expiring *hrtimer* is simply retrieved by checking the first timer in the time ordered red-black tree.

On the other hand, the framework will deliver all the necessary clock event source mechanisms to reprogram the next event interrupt and enable a clean, non-intrusive, out of the box, solution once an architecture has been converted to use the framework components.

The clock event functionalities necessary for dynamic tick implementations are available whether the high resolution timer functionality is enabled or not. The framework code takes care of those use cases already.

With the integration of dynamic ticks the transformation of the Linux time related subsystems will become complete, as illustrated in Figure 6.

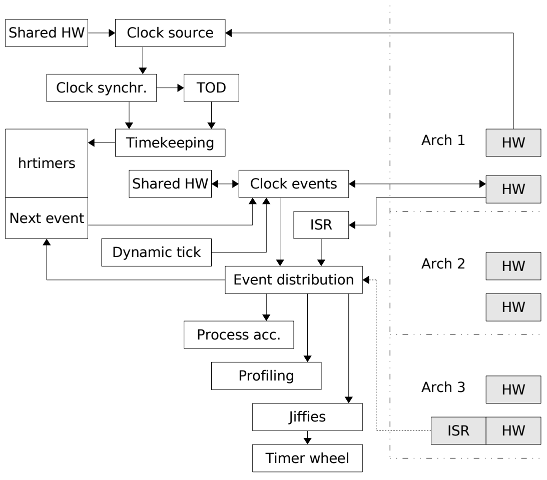


Figure 6: Transformed Linux Time Subsystem

10 Conclusion

The existing parts and pieces of the overall solution have proved that a generic solution for high resolution timers and dynamic tick is feasible and provides a valuable benefit for the Linux kernel.

Although most of the components have been tested extensively in the high resolution timer patch and the real-time preemption patch there is still a way to go until a final inclusion into the mainline kernel can be considered.

In general this can only be achieved by a step by step conversion of functional units and architectures. The framework code itself is almost self contained so a not converted architecture should not have any impacts.

We believe that we provided a clear vision of the overall solution and we hope that more developers get interested and help to bring this further in the near future.

10.1 Acknowledgments

We sincerely thank all those people who helped us to develop this solution. Help has been provided in form of code contributions, code reviews, testing, discussion, and suggestions. Especially we want to thank Ingo Molnar, John Stultz, George Anzinger, Roman Zippel, Andrew Morton, Steven Rostedt and Benedikt Spranger. A special thank you goes to Jonathan Corbet who wrote some excellent articles about hrtimers (and the previous ktimers) implementation [2, 3].

References

[1] George Anzinger and Monta Vista. High resolution timers home page.

`http://high-res-timers.sourceforge.net`.

[2] J. Corbet. Lwn article: A new approach to kernel timers. `http://lwn.net/Articles/152436`.

[3] J. Corbet. Lwn article: The high resolution timer api. `http://lwn.net/Articles/167897`.

[4] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the shelf hardware and free software. In 4th *Real-Time Technology and Applications Symposium*, Denver, June 1998.

[5] J. Stulz. We are not getting any younger: A new approach to timekeeping and timers. In *Ottawa Lnux Symposium*, Ottawa, Ontario, Canada, July 2005.

Making Applications Mobile Under Linux

Cédric Le Goater, Daniel Lezcano, Clément Calmels
IBM France
{clg, dlezcano, clement.calmels}@fr.ibm.com

Dave Hansen, Serge E. Hallyn
IBM Linux Technology Center
{haveblue, serue}@us.ibm.com

Hubertus Franke
IBM T.J. Watson Research Center
frankeh@watson.ibm.com

Abstract

Application mobility has been an operating system research topic for many years. Many approaches have been tried and solutions are found across the industry. However, performance remains the main issue and all the efforts are now focused on performant solutions. In this paper, we will discuss a prototype which minimizes the overhead at runtime and the amount of application state. We will examine constraints and requirements to enhance performance. Finally, we will discuss features and enhancements in the Linux kernel needed to implement migration of applications.

1 Introduction and Motivation

Applications increasingly run for longer periods of time and build more context over time as well. Recovering that context can be time consuming, depending on the application, and usually requires that the application be re-run from the beginning to reconstruct its context. A few applications now provide the ability to checkpoint their data or context to a file, enabling that application to be restarted later in the case of a failure, a system upgrade, or a need to redeploy hardware resources. This ability to checkpoint context is most common in what is referred to as the High Performance Computing (HPC) environment, which is often composed of large numbers of computers working on a distributed, long running computation. The applications often run for days or weeks at a time, some even as long as a year.

Even outside the HPC arena, there are many applications which have long start up times, long periods of processing configuration files, pre-computing information and so on. Historically, emacs was built with a script which included `undump`—the ability to checkpoint the full state of emacs into a binary which could then be started much more quickly. Some enterprise class applications have thirty minute start up times, and those applications continue to build complex context as they continue to run.

Increasingly we as users tend to expect that our applications will perform quickly, start quickly, re-start quickly on a failure, and be always available. However, we also expect to be able to upgrade our operating system, apply security fixes, add components, memory, sometimes even processing power without losing all of the context that our applications have acquired.

This paper discusses a generic mechanism for saving the state of an application at any point, with the ability to later restart that application exactly where it left off. This ability to save status and restart an application is typically referred to as checkpoint/restart, abbreviated throughout as CPR. This paper focuses on the key areas for allowing applications to be virtualized, simplifying the ability to checkpoint and later restart an application. Further, the technologies covered here would allow applications to potentially be restarted on a different operating system image than the one from which it was checkpointed. This provides the ability to move an application (or even a set of applications) dynamically from one machine or virtual operating system image to another.

Once the fundamental mechanisms are in place, this technology can be used for such more advanced capabilities such as checkpointing a cluster wide application—in other words synchronizing and stopping a coordinated, distributed applications, and restarting them. Cluster wide CPR would allow a site administrator to install a security update or perform scheduled maintainance on the entire cluster without impacting the application running.

Also, CPR would enable applications to be moved from host to host depending on system load. For instance, an overloaded machine could have its workload rebalanced by moving an application set from one machine to another that is otherwise underutilized. Or, several systems which are underloaded could have their applications consolidated to a single machine. CPR plus migration will henceforth be referred to as CPRM.

Most of the capabilities we've highlighted here are best enabled via application virtualization. Application virtualization is a means of abstracting, or virtualizing, the software resources of the system. These include such things as process id's, IPC ids, network connections, memory mappings, etc. It is also a means to contain and isolate resources required by the application to enable its mobility. Compared to the virtual machine approach, application virtualization approach minimizes the state of the application to be transferred and also allows for a higher degree of resource sharing between applications. On the other hand, it has limited fault containment, when compared to the virtual machine approach.

We built a prototype, called MCR, by modifying the Linux kernel and creating such a layer of containment. They are various other projects with similar goals, for instance VServer [8] and OpenVZ [7] and dated Linux implementation of BSD Jails [4]. In this paper, we will describe our experiences from implementing MCR and examine the many communalities of these projects.

2 Related Work in CPR

CPR is theoretically simple. Stop execution of the task and store the state of all memory, registers, and other resources. To restart, reload the executable image, load the state saved during the checkpoint, and restart execution at the location indicated by the instruction pointer register. In practice, complications arise due to issues like inter-processes sharing, security implications, and the ways that the kernel transparently manages resources. This section groups some of the existing solutions and reviews their shortcomings.

Virtual machines control the entire system state, making CPR easy to implement. The state of all memory and resources can simply be stored into a file, and recreated by the machine emulator or the operating system itself. Indeed, the two most commonly mentioned VMs, VMware [9] and Xen [10], both enable live migration of their guest operating systems. The drawbacks of CPRM of an entire virtual machine is the increased overhead of dealing with all resources defining the VM. This can make the approach unsuitable for load balancing applications, since a requirement to add the overhead of a full VM and associated daemons to each migrateable application can have tremendous performance implications. This issue is further explored in Section 6.

A more lighter weight CPRM approach can be achieved by isolating applications, which is predicated on the safe and proper isolation and migration of its underlying resources. In general, we look at these isolated and migrateable units as containers around the relevant processes and resources. We distinguish conceptually *system containers*, such as VServer [8] or OpenVZ [7], and *application containers*, such as Zap [12] and our own prototype MCR. Since containers share a single OS instance, many resources provided by the OS must be specially isolated. These issues are discussed in detail in Section 5. Common to both container approaches is their requirement to be able to CPR an isolated set of individual resources.

For many applications CPR can be completely achieved from user space. An example implementation is `ckpt` [5]. Ckpt teaches applications to checkpoint themselves in response to a signal by either preloading a library, or injecting code after application startup. The new code, when triggered, writes out a new executable file. This executable reloads the application and resets its state before continuing execution where it left off. Since this method is implemented with no help from the kernel, there is state which cannot easily be stored, such as pending signals, or recreated, such as a process' original process id. This method is also potentially very inefficient as described in Section 5.2. The user space approaches also fall short by requiring applications to be rewritten and by exhibiting poor resource sharing.

CPR becomes much easier given some help from the kernel. Kernel-based CPR solutions include `zap` [12], `crak` [2], and our MCR prototype. We will be analyzing MCR in detail in Section 4, followed by the requirements for a consolidated application virtualization and migration kernel approach.

3 Concepts and principles

A user application is a set of resources—tasks, files, memory, IPC objects, etc.—that are aggregated to provide some features. The general concept behind CPR is to freeze the application and save all its state (in both kernel and user spaces) so that it can be resumed later, possibly on another host. Doing this transparently without any modification to the application code is quite an easy task, as long as you maintain a single strong requirement: *consistency*.

3.1 Consistency

The kernel ensures *consistency* for each resource's internal state and also provides system identifiers to user space to manipulate these resources. These identifiers are part of the application state, and as such are critical to the application's correct behavior. For example, a process waiting for a child will use the known process id of that child. If you were to resume a checkpointed application, you would recreate the child's pid to ensure that the parent would wait on the correct process. Unfortunately, Linux does not provide such control on how pids, or many other system identifiers, are associated with resources.

An interesting approach to this problem is virtualization: a resource can be associated with a supplementary virtual system identifier for user space. The kernel can maintain associations between virtual and system identifiers and offer interfaces to control the way virtual identifiers are assigned. This makes it possible to change the underlying resource and its system identifier without changing the virtual identifier known by the application. A direct side effect is that such virtualized applications can be confused by virtual identifier collisions if they are not separated from one another. These conflicts can be avoided if the virtualization is implemented with resource containment features. For example, `/proc` should only export virtual pid entries for processes in the same virtualized container as the reading process.

3.2 Process subsystem

Processes are the essential resources. They offer many features and are involved in many relationships, such as parent, thread group, and process group. The pid is involved in many system calls and regular UNIX features such as session management and shell job control. Correct virtualization must address the whole picture to preserve existing semantics. For example, if we want to run multiple applications in different containers from the same login session, we will also want to keep the same system session identifier for the ancestor of the container so as to still benefit from the regular session cleanup mechanism. The consequence is that we need a system pid to be virtualized multiple times in different containers. This means that any kernel code dealing with pids that are copied to/from user space must be patched to provide containment and choose the correct virtualization space according to the implied context.

3.3 Filesystem and Devices

A filesystem may be divided into two parts. On one hand, global entries that are visible from all containers like `/usr`. On the other hand, local entries that may be specific to the containers like `/tmp` and of course `/proc`. `/proc` virtualization for numerical process entries is quite straightforward: simply generate a filename out of the virtual pid. Tagging the resultant `dentries` with a container id makes it possible to support conflicting names in the directory name cache and to filter out unrelated entries during `readdir()`.

The same tagging mechanism can be applied using files attributes for instance. Some user level administration commands can be used by the system administrator to keep track of files created by the containers. Devices files can be tagged to be visible and usable by dedicated containers. And of course, the `mknod` system call should fail on containers or be restricted to a minimal usage.

3.4 Network

If the application is network oriented, the migration is more complex because resources are seen from outside the container, such as IP addresses, communication ports, and all the underlying data related to the TCP protocol. Migration needs to take all these resources into account, including in-flight messages.

The IP address assigned to a source host should be recreated on the target host during the migration. This mobile IP is the foundation of the migration, but adds a constraint on the network. We will need to stay on the same network.

The migration of an application will be possible only if the communication channels are clearly isolated. The connections and the data associated with each application should be identified. To ensure such a containment, we need to isolate the network interfaces.

We will need to freeze the TCP layer before the checkpoint, to make sure the state of the peers is consistent with the snapshot. To do this, we will block network traffic for both incoming and outgoing packets. The application will be migrated immediately after the checkpoint and all the network resources related to the container will be cleaned up.

4 Design Overview

We have designed and implemented MCR, a lightweight application oriented container which supports mobility. It is discussed here because it is one of a few implementations which are relatively complete. It provides an excellent view on what issues and complexities arise. However, from our prototype work, we have concluded that certain functionality implemented in user space in MCR is best supported by the kernel itself.

An important idea behind the design of MCR is that it is kernel-friendly and does not do everything in kernel space. A balance needed to be struck between striving towards a minimal kernel impact to facilitate proper forward ports and ensuring that functional correctness and acceptable performance is achieved. This means using available kernel features and mechanisms where possible and not violating important principles which ensure that user space applications work properly.

With that principle in mind, CPR from user space makes your life much easier. It also enables some nifty and useful extensions like distributed CPR and system management.

4.1 Architecture

The following section provides an overview of the MCR architecture (Figure 1). It relies on a set of user level utilities which control the container: creation, checkpoint, restart, etc. New features in the kernel and a kernel module are required today to enable the container and the CPR features.

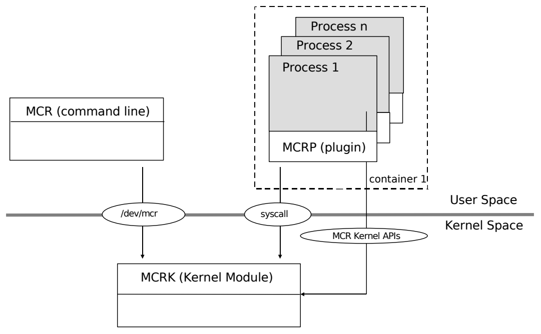


Figure 1: MCR architecture

The CPR of the container is not handled by one component. It is *distributed* across the 3 components of MCR depending on the locality of the resourse to checkpoint. The user level utility `mcr` (Section 4.1.2) invokes and orchestrates the overall checkpoint of a container. It also is in charge of checkpointing the resources which are global to a container, like SYSV shared memory for instance. The user level plugin `mcrp` (Section 4.1.4) checkpoints the resources at the process level, memory, and at the thread level, signals and cpu state. Both rely on the kernel module `mcrk` (Section 4.1.3) to access kernel internals.

4.1.1 Kernel enhancements and API

Despite a user space-oriented approach, the kernel still requires modifications in order to support CPR. But, surprisingly, it may no be as much as one might expect. There are three high level needs:

The biggest challenge is to build a container for the application. The aim here is neither security nor resource containment, but making sure that the snapshot taken at checkpoint time is *consistent*. To that end we need a container much like the VServer[8] context. This will isolate and identify all kernel resources and objects used by an application. This kernel feature is not only a key requirement to application mobility, but also for other frameworks in the security and resource management domains. The container will also virtualize system identifiers to make sure that the resources used by the application do not overlap with other containers. This includes resources such as processes IDs, threads IDs, SysV IPC IDs, UTS names, and IP addresses, among others.

The second need regards freezing a container. Today, we use the `SIGSTOP` signal to freeze all running tasks. This gives valid results, but for upstream kernel development we would prefer to use a container version of the `refrigerator()` service from `swsusp`. `swsusp` uses fake signals to freeze nearly all kernel tasks before dumping the memory.

Finally, MCR exposes the internals of different Linux subsystems and provides new services to *get* and *set* their state. These interfaces are by necessity very intrusive, and expose internal state. For an upstream migration solution, using a `/proc` or `/sysfs` interface which exposes more selective data would be more appropriate.

4.1.2 User level utilities

Applications are made mobile simply by being started under `mcr-execute`. The container is created before the `exec()` of the command starting the application. It is maintained around the application until the last process dies.

`mcr-checkpoint` and `mcr-restart` invoke and orchestrate the overall checkpoint or restart of a container. These commands also perform the *get* and *set* of the resources which are global to a container, as opposed to those local to a single process within the container. For instance, this is where the SYSV IPC and file descriptors are handled. They rely on the kernel module (Section 4.1.3) to manage the container and access kernel internals.

4.1.3 Kernel module

The kernel module is the container manager in terms of resource usage and resource virtualization. It maintains a real time definition of the container view around the application which ensures that a checkpoint will be consistent at any time.

It is in charge of the global synchronization of the container during the CPR sequence. It freezes all tasks running in the container, and maps into the process a user level plugin (see Section 4.1.4). It provides the synchronization barriers which unrolls the full sequence of the checkpoint before letting each process resume its execution.

It also acts as a *proxy* to capture the states which cannot be captured directly from user space. Internal states handled by this module include, for example, the process' memory page mapping, socket buffers, clone flags, and AIO states.

4.1.4 User level plugin

When a checkpoint or a restart of a container is invoked, the kernel module maps a plugin into each process of the container. This plugin is run in the process context and is removed after completion of the checkpoint. It serves 2 purposes. The first is synchronization, which it orchestrates with the help of the kernel module. Secondly, it performs *get* and *set* of states which can be handled from user space using standard syscalls. Such states include sigactions, memory mapping, and rlimits.

When all threads of a process enter the plugin, a *master* thread, not necessarily the main thread, is elected to handle the checkpoint of the resources at process level. The other threads only checkpoint the resources at thread level, like cpu state.

4.2 Linux subsystems CPR

The following subsections describe the checkpoint and the restart of the essential resources without doing a deep dive in all the issues which need to be addressed. The following section 5 will delve deeper into selected issues for the interested reader.

4.2.1 CPU state

Checkpointing the cpu state is indirectly done by the kernel because the checkpoint is signal oriented: it is saved by the kernel on the top of the stack before the signal handler is called. This stack is then saved with the rest of the memory. At restart, the kernel will restore the cpu state in `sigreturn()` when it jumps out of the signal handler.

4.2.2 Memory

The memory mapping is checkpointed from the process context, parsing `/proc/self/maps`. However, some `vm_area` flags (i.e. `MAP_GROWSDOWN`) are not exposed through the `/proc` file system. The latter are read from the kernel using the kernel module. The same method is used to retrieve the list of the mapped pages for each `vm_area` and reduce significantly the size of the snapshot.

Special pages related to POSIX shared memory and POSIX semaphores are detected and skipped. They are handled by the checkpoint of resources global to a container.

4.2.3 Signals

Signal handlers are registered using `sigaction()` and called when the process gets a signal. They are checkpointed and restarted using the same service in the process context. Signals can be sent to the process or directly to an invidual thread using the `tkill()` syscall. In the former, the signal goes into a shared sigpending queue, where any thread can be selected to handle it. In the latter, the signal goes to a thread private sigpending queue. To guarantee correct signal ordering, these queues must be checkpointed and restored separately using a dedicated kernel service.

4.2.4 Process hierarchy

The relationships between processes must be preserved across CPR sequences. Groups and sessions leaders are detected and taken into account. At checkpoint, each process and thread stores in the snapshot its execution command using `/proc/self/exe` and its pid and ppid using `getpid()` and `getppid()`. Threads also need to save their tid, ptid, and their stack frame. At restart time, the processes are recreated by `execve()` and immediately killed with the checkpoint signal. Each process then jumps into the user level plugin (See Section 4.1.4) and spawns its children. Each process also respawns its threads using `clone()`. The process tree is recreated recursively. On restart, attention must be paid to correctly setting the pid, pgid, tid, tgid for each newly created process and thread.

4.2.5 Interprocess communication

The contents and attributes of SYSV IPCs, and more recently POSIX IPCs, are checkpointed as resources global to a container, excepting semundos.

Most of the IPC resource checkpoint is done at the user level using standard system calls. For example, `mq_receive()` to drain all messages from a queue, and `mq_send()` to put them back into the queue. The two main drawbacks to such an approach are that access time to resources are altered and that the process must have read and write access to them. Some functionalities like `mq_notify()` are a bit trickier. In these cases, the kernel sends notification cookies using an `AF_NETLINK` socket which also needs to be checkpointed.

4.2.6 Threads

Every thread in a process shares the same memory, but has its own register set. The threads can dump themselves in user context by asking the kernel for their properties. At restart time, the main thread can read other threads' data back from the snapshot and respawn each with its original tid.

The thread local storage contains the thread specific information, data set by `pthread_setspecific()` and the `pthread_self()` pointer. On some architectures it is stored in a general purpose register. In that case it is already covered in the signal handler frame. But on some other architectures, like Intel, it is stored in a separate segment, and this segment mapping must be saved using a dedicated call to kernel.

4.2.7 Open files

File descriptors reference open files, which can be of any type, including regular files, pipes, sockets, FIFOs, and POSIX message queues.

CPR of file descriptors is not done entirely in the process context because they can be shared. Processes get their fd list by walking `/proc/self/fd`. They send this list, using ancillary messages, to a helper daemon running in the container during the checkpoint. Using the address of the `struct file` as a unique identifier, the daemon checkpoints the file descriptor only once per container, since two file descriptors pointing to the same opened file will have the same `struct file`.

File descriptors 0, 1, and 2 are considered special. We may not want to checkpoint or restart them if we do the restart on another login console for example. The file descriptors are tagged and bypassed at checkpoint time.

4.2.8 Asynchronous I/O

Asynchronous I/Os are difficult to handle by nature because they can not easily be frozen. The solution we found is to let the process reach a quiescence point where all AIOs have completed before checkpointing the memory. The other issue to cover is the ring buffer of completed events which is mapped in user space and filled by the kernel. This memory area needs to be mapped at the same address when the process is restarted. This requires a small patch to control the address used for the mapping.

5 Zooming in

This section will examine in more detail three major aspects of application mobility. The first topic covers a key requirement in process migration: the ability to restart a process keeping the same pid. Next we will discuss issues and solutions to VM migration, which has the biggest impact on performance. Finally, we will address network isolation and migration of live network communications.

5.1 Process Virtualization

A `pid` is a handle to a particular task or task group. Inside the kernel, a `pid` is dynamically assigned to a task at fork time. The relationship is recorded in the pid hash table ($pid \rightarrow task$) and remains in place until a task exits. For system calls that return a pid (e.g. `getpid()`), the pid is typically extracted straight out of the task structure. For system calls that utilize a user provided pid, the task associated with that pid is determined from the pid hash table. In addition various checks need to be performed that guarantee the isolation between users and system tasks (in particular during the `sys_kill()` call).

Because `pids` might be cached at the user level, processes should be restarted with their original pids. However, it is difficult if not impossible to ensure that the same `pid` will always be available upon restart of a checkpointed application, as another process could already have been started with this pid. Hence, `pids` need to be *virtualized*. Virtualization in this context can be and is interpreted in various manners. Ultimately the requirement, that an application consistently sees the same `pid` associated with a task (process/thread) across CPR, must be satisfied.

There are essentially three issues that need to be dealt with in any solution:

1. container init process visibility,
2. where in the kernel the virtualization interception will take place,
3. how the virtualization is maintained.

Particularly 1. is responsible for the non-trivial complexities of the various prototypes. It stems from the necessity to "rewrite" the pid relationships between the top process of a container (short `cinit`) and its parent. `cinit` essentially lives in both contexts, the creating container and the created container. The creating container requires a pid in its context for `cinit` to be able to deploy regular `wait()` semantics. At the same time, `cinit` must refer to its parent as the perceived system init process ($vpid = 1$).

5.1.1 Isolation

Various solutions have been proposed and implemented. Zap [12] intercepts and wraps all pid related system calls and virtualizes the pids in the interception layer through a $pid \leftrightarrow vpid$ lookup table associated with the caller's container either before and/or after calling the original syscall implementation with the real pids. The benefit of this approach is that the kernel does not need to be modified. However, the ability of overwriting the syscall table is not a direction Linux embraces.

MCR, presented in greater detail in Section 4, pushes the interception further down into the various syscalls itself, but also utilizes a *pid* $\leftrightarrow$ *vpid* lookup function. In general, the calling task provides the context for the lookup.

The isolation between containers is implemented in the lookup function. Tasks that are created inside a container are looked up through this function. For global tasks the `pid == vpid` holds. In both implementations the `vpid` is not explicitly stored with the task, but is determined through the *pid* $\leftrightarrow$vpid lookup each and every time. On restart tasks can be recreated through the `fork(); exec()` sequence and only the lookup table needs to record the different `pid`. The `cinit` parent problem mentioned earlier is solved by mapping `cinit` twice, in the created context as vpid=1 and in the creating container contexts with the assigned vpid. The lookup function is straight forward, essentially we need to ensure that we identify any `cinit` process and return the vpid/task associated with it relative to the provided container context.

The OpenVZ implementation [7] provides an interesting, yet worthwhile optimization that only requires a lookup for tasks that have been restarted. OpenVZ relies on the fact that tasks do have a unique pid when tasks are in their original incarnation (not yet C/R'd). The lookup function, which is called at the same code locations as the MCR implementation, hence only has to maintain the isolation property. In the case of a restarted task the uniqueness can no further be guaranteed, so the pid must be virtualized. Common to all three approaches is the fact that virtual pids are all relative to their respective containers and that they are translated into system-wide unique pids. The guts of the pidhash have not changed.

A different approach is taken by the namespace proposal [6]. Here, the container principle is driven further down into the kernel. The pidhash now is defined as a ($\{pid, container\} \rightarrow task$) function. The namespace approach naturally elevates the container as a first class kernel object. Hence minor changes were required to the pid allocation which maintains a pidmap for each and every namespace now. The benefit of this approach is that the code modifications clearly highlight the conditions where container boundaries need to be crossed, where in the earlier virtualization approach these crossings came implicitly through the results of the lookup function. On the other hand, the namespace approach needs special provisioning for the `cinit` problem. To maintain the ability to `wait()` on the cinit process from the cinit's parent (child_reaper), a `task->wid` is defined, that reflects the pid in the parent's context and on which the parent needs to wait. There is no clear recommendation between the namespace and virtualization approach that we want to give in this paper; both the OpenVZ and the namespace proposal are very promising.

5.1.2 CPR on Process Virtualization

The CPR of the Process Virtualization is straight forward in both cases. In the ZAP, MCR and OpenVZ case, the lookup table is recreated upon restart and populated with the vpid and the real pid translations, thus requiring the ability to select a specific vpid for a restarted process. Which real pid is chosen is irrelevant and is hence left to the pidmap management of the kernel. In the namespace approach since the pid selection is pushed into the kernel a function requires that a task can be forked at a specific pid with in a container's pidmap. Ultimately, both approaches are very similar to each other.

5.2 CPR on the Linux VM

At first glance, the mechanism for checkpointing a process's memory state is an easy task. The mechanism described in section 2 can be implemented with a simple ptrace.

This approach is completely in user space, so why is it not used in the MCR prototype, nor any commercial CPR systems? Or in other words, why do we need to push certain functionalities further down into the kernel?

5.2.1 Anonymous Memory

One of the simplest kind of memory to checkpoint is anonymous memory. It is never used outside the process in which it is allocated.

However, even this kind of memory would have serious issues with a ptrace approach.

When memory is mapped, the kernel does not fill it in at that time, but waits until it is used to populate it. Any user space program doing a checkpoint could potentially have to iterate over multiple gigabytes of sparse, entirely empty memory areas. While such an approach could consolidate such empty memory after the fact, simply iterating over it could be an incredibly significant resource drain.

The kernel has intimate knowledge of which memory areas actually contain memory, and can avoid such resource drains.

5.2.2 Shared Memory

The key to successful CPR is getting a consistent snapshot. If two interconnected processes are checkpointed at different times, they may become confused when restarted. Successful memory checkpointing requires a consistent quiescence of all tasks sharing data. This includes all shared memory areas and files.

5.2.3 Copy on Write

When a process forks, both the forker and the new child have exactly the same view of memory. The kernel gives both processes a read-only view into the same memory. Although not explicit, these memory areas are shared as long as neither process writes to the area.

The above proposed ptrace mechanism would be a very poor choice for any processes which have these copy-on-write areas. The areas have no practical bounds on their sizes, and are indistinguishable from normal, writable areas from the user's (and thus ptrace's) perspective.

Any mechanism utilizing the ptrace mechanism could potentially be forced to write out many, many copies of redundant data. This could be avoided with checksums, but it causes user space reconstruction of information about which the kernel already explicitly knows.

In addition, user space has no way of explicitly recreating these copy-on-write shared areas during a resume operation. The only mechanism is fork, which is an awfully blunt instrument by which to recreate an entire system full of processes sharing memory in this manner. The only alternative is restoring all processes and breaking any sharing that was occurring before the checkpoint. Breaking down any sharing is highly undesirable because it has the potential to greatly increase memory utilization.

5.2.4 Anonymous Shared

Anonymous shared memory is that which is shared, but has no backing in a file. In Linux there is no true anonymous shared memory.

The memory area is simply backed by a pseudo file on a ram-based filesystem. So, there is no disk backing, but there certainly is a file backing.

It can only be created by an `mmap()` call which uses the `MAP_SHARED` and `MAP_ANONYMOUS`. Such a mapping is unique to a single process and not truly shared. That is, until a `fork()`.

No running processes may attach to such memory because there is no handle by which to find or address it, neither does it have persistence. The pseudo-file is actually deleted, which creates a unique problem for the CPR system.

Since the "anonymous" file is mapped by some process, the entire addressable contents of the file can be recovered through the aforementioned ptrace mechanism. Upon resume, the "anonymous" areas can be written to a real file in the same ram-based filesystem. After all processes sharing the areas have recreated their references to the "anonymous" area, the file can be deleted, preserving the anonymous semantics. As long as the process performing the checkpoint has ptrace-like capabilities for all processes sharing the memory area, this should not be difficult to implement.

5.2.5 File-backed Shared

Shared memory backed by files is perhaps the simplest memory to checkpoint. As long as all dirty data has been written back, requiring filesystem consistency be kept between a checkpoint and restart is all that is required. This can be done completely from user space.

One issue is with deleted files. However, these can be treated in the same way as "anonymous" shared memory mentioned above.

5.2.6 File-backed Private

When an application wants a copy of a file to be mapped into memory, but does not want any changes reflected back on the disk, it will map the file `MAP_PRIVATE`.

These areas have the same issues as anonymous memory. Just like anonymous memory, separately checkpointing a page is only necessary after a write. When simply read, these areas exactly mirror contents on the disk and do not need to be treated differently from normal file-backed shared memory.

However, once a write occurs, these areas' treatment resembles that of anonymous memory. The contents of each area must be read and preserved. As with anonymous memory, user space has no detailed knowledge of specific pages having been written. It must simply assume that the entire area has changed, and must be checkpointed.

This assumption can, of course, be overridden by actually comparing the contents of memory with the contents of the disk, choosing not to explicitly write out any data which has not actually changed.

5.2.7 Using the Kernel

From the ptrace discussions above, it should be apparent that the he various kinds of memory mappings in Linux can be checkpointed from userspace while preserve many of their important pre-checkpoint attributes. However, it should now be apparent that user space lacks the detailed knowledge to do these operations efficiently.

The kernel has exact knowledge of exactly which pages have been allocated and populated. Our MCR prototype uses this information to efficiently create memory snapshots.

It walks the pagetables of each memory area, and marks for checkpoint only those pages which actually have contents, and have been touched by the process being checkpointed. For instance, it records the fact that "page 14" in a memory area has contents. This solves the issues with sparsely populated anonymous and private file-backed memory areas, because it accurately records the process's actual use of the memory.

However, it misses two key points: the simple presence of a page's mapping in the page tables does not indicate whether its contents exactly mirror those on the disk.

This is an issue for efficiently checkpointing the file-backed private areas because the page may be mapped, but it may be either a page which has been only read, or one to which a write has occurred. To properly distinguish between the two, the `PageMappedToDisk()` flag must be checked.

5.2.8 File-backed Remapped

Assume that a file containing two pages worth of data is `mmap()`ed. It is mapped from the beginning of the file through the end. One would assume that the first page of that mapping would contain the first page of data from the disk. By default, this is the behavior. But, Linux contains a feature which invalidates this assumption: `remap_file_pages()`.

That system call allows a user to remap a memory area's contents such that the n^{th} page of a mapping does not correspond to the n^{th} page on the disk. The only place in which the information about the mapping is stored is in the pagetables. In addition, the presence of one of these areas is not openly available to user space.

Our user space ptrace mechanism could likely detect these situations by double-checking that each page in a file-backed memory area is truly backed by the contents on the disk, but that would be an enormous undertaking. In addition, it would not be a complete solution because two different pages in the file could contain the same data. Userspace would have absolutely no way to uniquely identify the position of a page in a file, simply given that page's contents.

This means that the MCR implementation is incomplete, at least in regards to any memory area to which `remap_file_pages()` has been applied.

5.2.9 Implementation Proposal

Any effective and efficient checkpoint mechanism must implement, at the least:

1. Detection and preservation of sharing of file-backed and other shared memory areas for both efficiency and correctness.
2. Efficient handling of sparse files and untouched anonymous areas.
3. Lower-level visibility than simply the file and contents for `remap_file_pages()` compatibility (such as effective page table contents).

There is one mechanism in the kernel today which deals with all these things: the swap code. It does not attempt to swap out areas which are file backed, or sparse areas which have not been populated. It also correctly handles the nonlinear memory areas from `remap_file_pages()`.

We propose that the checkpointing of a process's memory could largely be done with a synthetic swap file used only by that container.

This swap file, along with the contents of the pagetables of the checkpointed processes, could completely reconstruct the contents of a process' memory. The process of checkpointing a container could become very similar to the operation which `swsusp` performs on an entire system.

The swap code also has a feature which makes it very attractive to CPR: the swap cache. The swap cache allows a page to be both mapped into memory **and** currently written out to swap space. The caveat is that, if there is a write to the page, the on-disk copy must be thrown away.

Memory which is very rarely written to, such as the file-backed private memory used in the jump table in dynamically linked libraries, has the most to gain from the swap cache. Users of this memory can run unimpeded, even during a checkpoint operation, as long as they do not perform writes.

Just as has been done with other live cross-system migration[11] systems, the process of moving the data across can be iterative. First, copy data in several passes until, despite the efforts to swap them out, the working set size of the applications ceases to decrease.

The application-level approach has the potential to be at least marginally faster than the whole-system migration because it is **only** concerned with application data. Xen must deal with the kernel's working set in addition to the application. This **must** increase the amount of data which must be migrated, and thus **must** increase the potential downtime during a migration.

5.3 Migrating Sockets

In Section 3.4, the needs for a network migration were roughly defined. This section focuses on the four essential networking components required for container migration: network isolation, network quiescent points, network state access for a CPR, and network resource cleanup.

5.3.1 Network isolation

The network interface isolation consists of selectively revealing network interfaces to containers. These can be either physical or aliased interfaces. Aliased interfaces are more flexible for managing the network in the containers because different IP addresses can be assigned to different containers with the same physical network interface. The `net_device` and `in_ifaddr` structures have been modified to store a list of the containers which may view the interface.

The isolation ensures that each container uses its own IP address. But any return packet must also go to the right interface. If a container connects to a peer without specifying the source address, the system is free to assign a source address owned by an another container. This must be avoided. The `tcp_v4_connect()` `udp_sendmsg()` functions are modified in order to choose a source address associated with an interface visible from the source container.

The network isolation ensures that network traffic is dispatched to the right container. Therefore it becomes quite easy to drop the traffic for any specific container.

5.3.2 Reaching a quiescent point

As the processes need to reach a quiescent point in order to stop their activities, the network must reach this same point in order to retrieve network resource states at a fixed moment.

This point is reached by blocking the network traffic for a specified container. The filtering mechanism relies on netfilter hooks. Every packet is contained in a `struct skbuff`. This structure has a link to the `struct sock` connection which has a record of the owner container. Using this mechanism, packets related to a container being checkpointed can be identified and dropped.

For dropping packets, iptables is not directly suitable because each drop rule returns a `NF_DROP`, which interacts with the TCP stack. But, we need the TCP stack to be frozen for our container. So a kernel module has been implemented which drops the packets but returns `NF_STOLEN` instead of `NF_DROP`.

This of course relies on the TCP protocol's retransmission of the lost packets. However, traffic blocking has a drawback: if the time needed for the migration is too large, the connections on the peers will be broken. The same will occur if the TCP keep alive time is too small. This encourages any implementation to have a very short downtime during a migration. However, note that, when both sides of a connection are checkpointed simultaneously, there are no problems with TCP timeouts. In that case the restart could occurs years later.

5.3.3 Socket CPR

Now that we have successfully blocked the traffic and frozen the TCP state, the CPR can actually be performed.

Retrieving information on UDP sockets is straightforward. The protocol control block is simple and the queues can be dropped because UDP communication is not reliable. Retrieving a TCP socket is more complex. The TCP sockets are classified into two groups: `SS_UNCONNECTED` and `SS_CONNECTED`. The former have little information to retrieve because the PCB (Protocol Control Block) is not used and the send/receive queues are empty. The latter have more information to be checkpointed, such as information related to the socket, the PCB, and the send/receive queues. Minisocks and the orphan sockets also fall in the connected category.

A socket can be retrieved from `/proc` because the file descriptors related to the current process are listed. The `getpeername()`, `getsockname()` and `getsockopt()` can be directly used with the file descriptors. However, some information is not accessible from user space, particularly the list of the minisocks and the orphaned sockets, because no fd is associated with them. The PCB is also unaccessible because it is an internal kernel structure. MCR adds several accessors to the kernel internals to retrieve this missing information.

The PCB is not completely checkpointed and restored because there is a set of fields which need to be modified by the kernel itself. For example, the round time trip values.

5.3.4 Socket Cleanup

When the container is migrated, the local network resources remaining in the source host should be cleaned up in order to avoid duplicate resources on the network. This is done using the container identifier. The IP addresses can not be used to find connections related to a container because if several interconnected containers are running on the same machine, there is no way to find the connection owner.

5.3.5 CPR Dynamics

The fundamentals for the migration have been described in the previous sections. Figure 2 illustrates how they are used to move network resources from one machine to an another.

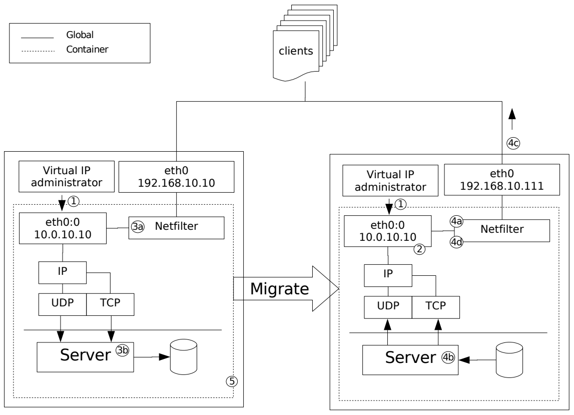


Figure 2: Migrating network resources

1. Creation

   A network administration component is launched; it creates an aliased interface and assigns it to a container.

2. Running

   The IP address associated with the aliased interface is the only one seen by the applications inside the container.

3. Checkpoint

   (a) All the traffic related to the aliased interface assigned to the container is blocked.

   (b) The network resources are retrieved for each kind of socket and saved: addresses, ports, multicast groups, socket options, PCB, in-flight data and listening points.

4. Restart

   (a) The traffic is blocked

   (b) The network resources are set from the file to the system.

   (c) An ARP (adress request package) response is sent to the network in order to boost up and ensure correct *mac* $\leftrightarrow$ *ip* address association.

   (d) The traffic is unblocked.

5. Destruction

   The traffic is blocked, the aliased interface is destroyed and the sockets related to the container are removed from the system.

6 What is the cost?

This section presents an overview of the cost of virtualization in different frameworks. We have focused on VServer, OpenVZ, and our own prototype MCR, which are all lightweight containers. We have also included Xen, when possible, as a point of reference in the field of full machine virtualization.

The first set of tests assesses the virtualization overhead on a single container for each above mentioned solution. The second measures scalability of each solution by measuring the impact of idle containers on one active container. The last set provides performance measures of the MCR CPR functionality with a real world application.

6.1 Virtualization overhead

At the time of this writing, no single kernel version was supported by each of VServer, OpenVZ, and MCR. Furthermore, patches against newer kernel versions come out faster than we can collect results, and clearly by the time of publication the patches and base kernel used in testing will be outdated anyway. Hence for each virtualization implementation we present results normalized against results obtained from the same version vanilla kernel.

6.1.1 Virtualization overhead inside a container

The following tests were made on quad PIII 700MHz running Debian Sarge using the following versions:

- VServer version vs2.0.2rc9 on a 2.6.15.4 kernel with `util-vserver` version 0.30.210
- MCR version 2.5.1 on a 2.6.15 kernel

We used *dbench* to measure filesystem load, *LMbench* for microbenchmarks, and a kernel build test for a generic macro benchmark. Each test was executed inside a container, with only one container created.

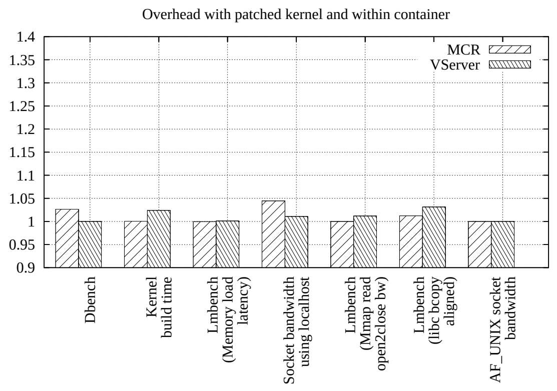


Figure 3: Various tests inside a container

The results shown in figure 3 demonstrate that the overhead is hardly measurable. OpenVZ, being a full virtualized server, was not taken into account.

6.1.2 Virtualization overhead within a virtual server

The next set of tests were run in a full virtual server rather than a simple container. For these tests, the nodes used were 64bit dual Xeon 2.8GHz (4 threads/2 real processors). Nodes were equipped with a 25P3495a IBM disk (SATA disk drive) and a Tigon3 gigabit ethernet adapter. The host nodes were running RHEL AS 4 update 1 and all guest servers were running Debian Sarge. We ran tbench and a 2.6.15.6 kernel build test in three environments: on the system running the vanilla kernel, on the host system running the patched kernel, and inside a virtual server (or guest system). The kernel build test was done with warmed up cache.

- VServer version 2.1.0 on a 2.6.14.4 kernel with `util-vserver` version 0.30.209
- OpenVZ version 022stab064 on a 2.6.8 kernel with `vzctl` utilities version 2.7.0-26
- Xen version 3.0.1 on a 2.6.12.6 kernel

The results are shown in the figures 4 and 5.

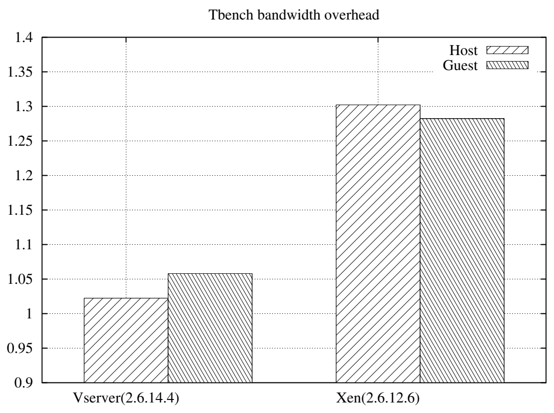


Figure 4: tbench results regarding a vanilla kernel

The OpenVZ virtual server did not survive all the tests. tbench looped forever and the kernel build test failed with a virtual memory allocation error. As expected, lightweight containers outperform a virtual machine. Considering the

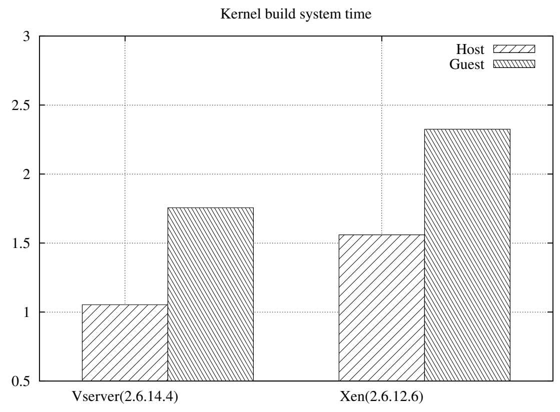


Figure 5: System time of a kernel build regarding a vanilla kernel

level of containment provided by Xen and the configuration of the domain, using file-backed virtual block device, Xen also behaved quite well. It would surely have better results with a LVM-backed VBD.

6.2 Resource requirement

Using the same tests, we have studied how performance is impacted when the number of containers increases. To do so, we have continously added idle containers to the system and recorded the application performance of the reference test in the presence of an increasing number of idle containers. This gives some insight in the resource consumption of the various virtualization techniques and its impact on application performance. This set of tests compared:

- VServer version 2.1.0 on a 2.6.14.4 kernel with `util-vserver` version 0.30.209
- OpenVZ version 022stab064 on a 2.6.8 kernel with `vzctl` utilities version 2.7.0-26
- Xen version 3.0.1 on a 2.6.12.6 kernel
- MCR version 2.5.1 on a 2.6.15 kernel

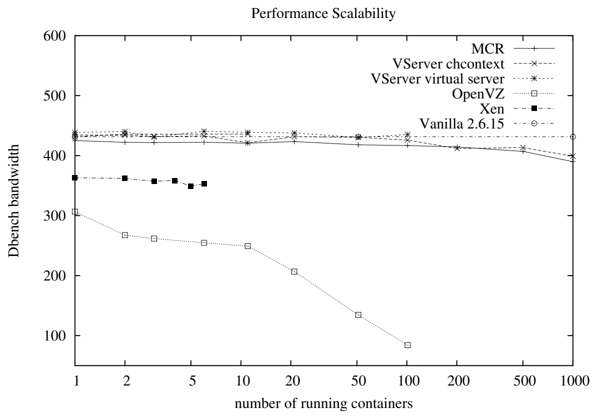


Figure 6: dbench performance with an increasing number of containers

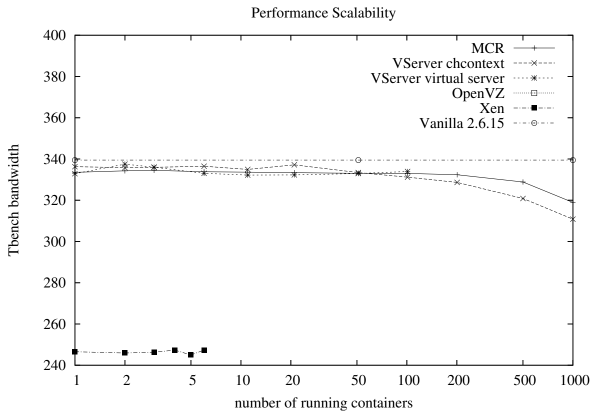


Figure 7: tbench performance with an increasing number of containers

The results are shown in the figures 6, 7, and 8. Lightweight containers are not really impacted by the number of idle containers. Xen overhead is still very reasonable but the number of simultaneous domains we were able to run stably was quite low. This issue is a bug in current Xen, and is expected to be solved. OpenVZ performance was poor again and did not survive the tbench test not the kernel build. The tests should definitely be rerun with a newer version.

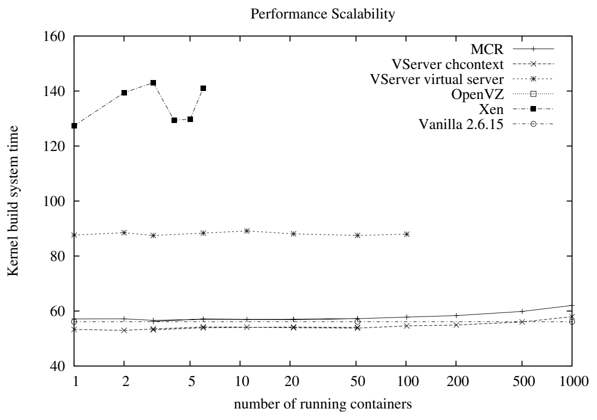


Figure 8: Kernel build time with an increasing number of containers

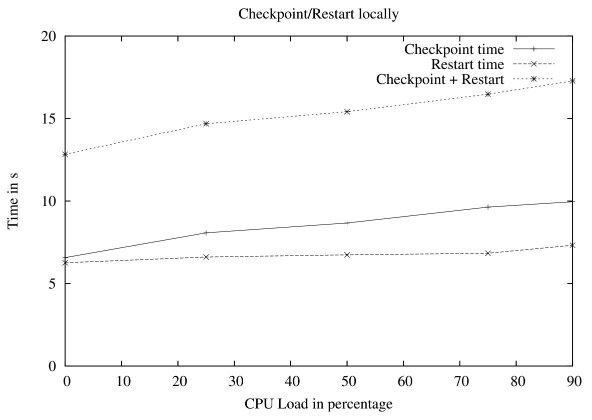


Figure 9: Oracle CPR time under load on local disk

6.3 Migration performance

To illustrate the cost of a migration, we have set up a simple test case with Oracle (`http://www.oracle.com/index.html`) and Dots, a database benchmark coming from the Linux Test Project (`http://ltp.sourceforge.net/`). The nodes used in our test were dual Xeon 2.4GHz HT (4 cpus/2 real processors). Nodes were equipped with a ST380011A (ATA disk drive) and a Tigon3 gigabit ethernet adapter. Theses nodes were running a RHEL AS 4 update 1 with a patched 2.6.9-11.EL kernel. We used Oracle version 9.2.0.1.0 running under MCR 2.5.1 on one node and Dots version 1.1.1 running on another node (nodes are linked by a gigabit switch). We measured the duration of checkpoint, the duration of restart and the size of the resulting snapshot with different Dots cpu workloads: no load, 25%, 50%, 75%, and 100% cpu load. The results are shown in Figures 9 and 10.

The duration of the checkpoint is not impacted by the load but is directly correlated to the size of the snapshot (real memory size). Using a swap file dedicated to a container and incremental checkpoints in that swap file (Section 5.2) should improve dramatically the checkpoint time. It will also improve service downtime when the application is migated from a node to another.

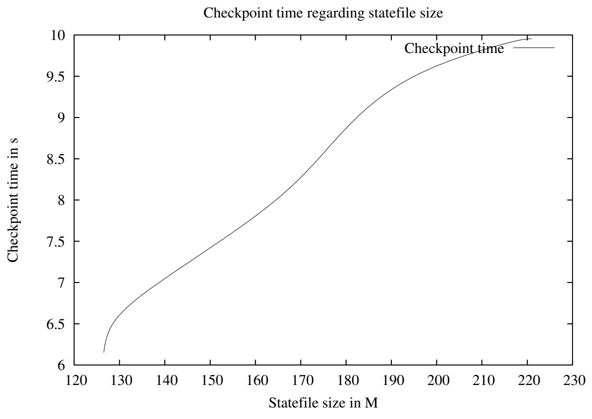


Figure 10: Time of checkpoint according to the snapshot size

At the time we wrote the paper, we were not able to run the same test with Xen, their migration framework not yet being available.

7 Conclusion

We have presented in this paper a motivation for application mobility as an alternative to the heavier virtual machine approach. We discussed our prototype of application mobility using the simple CPR approach. This prototype helped us to identify issues and work on solutions that would bring useful features to the Linux kernel. These features are isolation of resources through containers, virtualization of resources and CPR of the kernel subsystem to provide mobility. We also went through the various other alternatives projects in that are currently persued within community and exemplified the many communalities and currents in this domain. We believe the time has come to consolidate these efforts and drive the necessary requirements into the kernel. These are the necessary steps that will lead us to live migration of applications as a native kernel feature on top of containers.

8 Acknowledgments

First all, we would like to thank all the former Meiosys team who developed the initial MCR prototype, Frédéric Barrat, Laurent Dufour, Gregory Kurz, Laurent Meyer, and François Richard. Without their work and expertise, there wouldn't be much to talk about. A very special thanks to Byoung-jip Kim from the Department of Computer Science at KAIST and Jong Hyuk Choi from the IBM T.J. Watson Research Center for their contribution to the s390 port, tests, benchmarks and this paper. And, most of all we need to thank Gerrit Huizenga for his patience and his encouragements. *Merci, DonkeySchön*!

9 Download

Patches, documentations and benchmark results will be available at `http://lxc.sf.net`.

References

[1] William R. Dieter and James E. Lumpp, Jr. *User-level Checkpointing for LinuxThreads Programs,* Department of Electrical and Computer Engineering, University of Kentucky

[2] Hua Zhong and Jason Nieh, *CRAK: Linux Checkpoint/Restart As a Kernel Module,* Department of Computer Science, Columbia University, Technical Report CUCS-014-01, November, 2001.

[3] Duell, J., Hargrove, P., and Roman., E. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart,* Berkeley Lab Technical Report (publication LBNL-54941).

[4] Poul-Henning Kamp and Robert N. M. Watson, *R. N. M. Jails: Confining the omnipotent root,* in Proc. 2nd Intl. SANE Conference (May, 2000).

[5] Victor Zandy, *Ckpt—A process checkpoint library,* `http://www.cs.wisc.edu/~zandy/ckpt/`.

[6] Eric Biederman, *Code to implement multiple instances of various linux namespaces,* `git://git.kernel.org/pub/scm/linux/kernel/git/ebiederm/linux-2.6-ns.git/`.

[7] SWSoft, *OpenVZ: Server Virtualization Open Source Project*, `http://openvz.org`, 2005.

[8] Jacques Gélinas, *Virtual private servers and security contexts*, `http://www.solucorp.qc.ca/miscprj/s_context.hc?prjstate=1&nodoc=0`, 2004.

[9] VMware Inc, *VMware*, `http://www.vmware.com/`, 2005.

[10] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer, *Xen and the art of virtualization,* in Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.

[11] Christopher Clark, Keir Fraser, Steven Hand, Jakob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield, *L*ive Migration of Virtual Machines, In Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05), May, 2005, Boston, MA.

[12] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. *The design and implementation of zap: A system for migrating computing environments,* in Proceedings of the 5th Usenix Symposium on Operating Systems Design and Implementation, pp. 361–376, December, 2002.

[13] Daniel Price and Andrew Tucker. "Solaris Zones: Operating System Support for Consolidating Commercial Workloads." From *Proceedings of the 18th Large Installation Systems Administration Conference* (USENIX LISA '04).

[14] Werner Almesberger, *Tcp Connection Passing,* `http://tcpcp.sourceforge.net`

Copyright © 2006 IBM.

This work represents the view of the authors and does not necessarily represent the view of IBM.

IBM and the IBM logo are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both. Other company, product, and service names may be trademarks or service marks of others. References in this publication to IBM products or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document is provided "AS IS," with no express or implied warranties. Use the information in this document at your own risk.

The What, The Why and the Where To of Anti-Fragmentation

Mel Gorman
IBM Corp. and Uni. of Limerick
mel@csn.ul.ie

Andy Whitcroft
LTC, IBM Corp.
andyw@uk.ibm.com

Abstract

Linux® uses a variant of the binary buddy allocator that is fast but suffers badly from external fragmentation and is unreliable for large contiguous allocations. We begin by introducing two cases where large contiguous regions are needed: the allocation of HugeTLB pages during the lifetime of the system and using memory hotplug to on-line and off-line memory on demand in support of changing loads. We also mention subsystems that may benefit from using contiguous groups of pages. We then describe two anti-fragmentation strategies, discuss their strengths and weaknesses and examine their implementations within the kernel. We cover the standardised tests, the metrics used, the system architectures tested in the evaluation of these strategies and conclude with an examination of their effectiveness at satisfying large allocations. We also look at a page reclamation strategy that is suited to freeing contiguous regions of pages and finish with a look at the future direction of anti-fragmentation and related work.

1 Introduction

The page allocator in any operating system is a critical component. It must be fast and have the ability to satisfy all requests to avoid subsystems building reserve page pools [4]. Linux uses a variant of the binary buddy allocator that is known to be fast in comparison to other allocator types [3] but behaves poorly in the face of fragmentation [5].

Fragmentation is a space-efficiency problem affecting all dynamic memory allocators and comes in two varieties; internal and external. Internal fragmentation occurs when a larger free block than necessary is granted for a request, such as allocating one entire page to satisfy a request for 32 bytes. Linux uses a slab allocator for small requests to address this issue. External fragmentation refers to the inability to satisfy an allocation because a suitably large block of memory is not free even though enough memory may be free overall [6]. Linux deals with external fragmentation by rarely requiring larger (*high order*) pages. Although this works well in general, Section 2 presents situations where it performs poorly.

To be clear, anti-fragmentation is not the same as defragmentation, which is a mechanism to reduce fragmentation by moving or reclaiming pages to have contiguous free space. Anti-fragmentation enables a system to conduct a partial defragmentation using the existing page reclamation mechanism. The remainder of this paper is arranged as described in the abstract.

2 Motivation for Low Fragmentation

HugeTLB pages are contiguous regions that match a large page size provided by an architecture, which is 1024 small pages on x86 and 4096 on PPC64. Use of these large pages reduces both expensive TLB misses [2] and the number of *Page Table Entries (PTEs)* required to map an area, thus increasing performance and reducing memory consumption. Linux keeps a HugeTLB freelist in the HugeTLB page pool. This pool is sized at boot time, which is a problem for workloads requiring different amounts of HugeTLB memory at different times. For example, workloads that use large in-memory data sets, such as X Windows, High-Performance Computing (HPC), many Java applications, and some desktop applications (e.g. Konqueror) require variable amounts of memory depending on the input data and type of usage. It is not possible to guess their needs at boot time. Instead it would be better to maintain low fragmentation so that their needs could be met as needed at run-time.

Contiguous regions are also required when a section of memory needs to be on-lined and then off-lined later. For example, a virtual machine running a service like a web server may require more memory due to a spike in usage, but later need to return the memory to the host. Some architectures can return memory to a *hypervisor* using a *balloon driver* but this only works when memory can be off-lined at the page granularity. The minimum sized region that can be off-lined is the same as the size of a *memory section* defined for the SPARSEMEM memory model. This model mandates that the memory section size be a power-of-two number of pages and the architecture selects a size within that constraint. On the PPC64, the minimum sized region of memory that can be off-lined is 16MiB which is the minimum size OpenFirmware uses for a *Logical Memory Block (LMB)*. On x86, the minimum sized region is 64MiB. This is the smallest DIMM size taken by the IBM xSeries® 445 which supports the memory hot-add feature. Low fragmentation increases the probability of finding regions large enough to off-line.

A third case where contiguous regions are desired, but not required, is for drivers that use DMA but do not support scatter/gather IO efficiently or do not have an IO-MMU available. These drivers must spend time breaking up the DMA request into page-sized units. Ideally, drivers could ask for a page-aligned block of memory and receive a list of large contiguous regions. With low fragmentation, the expectation is that the driver would have a better chance of getting one contiguous block and not need to break up the request.

3 External Fragmentation

The extent of fragmentation depends on the number of free blocks[1] in the system, their size and the size of the requested allocation. In this section, we define two metrics that are used to measure the ability of a system to satisfy an allocation and the degree of fragmentation.

We measure the fraction of available free memory that can be used to satisfy allocations of a specific size using an *unusable free space index*, F_u.

$$F_u(j) = \frac{TotalFree - \sum_{i=j}^{i=n} 2^i k_i}{TotalFree}$$

[1]A free block is a single contiguous region stored on a freelist. In rare cases with the buddy allocator, two free blocks are adjacent but not merged because they are not buddies.

where $TotalFree$ is the number of free pages, 2^n is the largest allocation that can be satisfied, j is the order of the desired allocation and k_i is the number of free page blocks of size 2^i. When $TotalFree$ is 0, we define F_u to be 1. A more traditional, if slightly inaccurate[2], view of fragmentation is available by multiplying $F_u(j)$ by 100. At 0, there is 0% fragmentation, at 1, there is 100% fragmentation, at 0.25, fragmentation is at 25% and 75% of available free memory can be used to satisfy a request for 2^j contiguous pages.

$F_u(j)$ can be calculated at any time, but external fragmentation is not important until an allocation fails [5] when $F_u(j)$ will be 1. We further define a *fragmentation index*, $F_i(j)$, which determines if the failure to allocate a contiguous block of 2^j pages is due to lack of memory or to external fragmentation. The higher the fragmentation of the system, the more free blocks there will be. At the time of failure, the ideal number of blocks shall be related to the size of the requested allocation. Hence, the index at the time of an allocation failure is

$$F_i(j) = 1 - \frac{TotalFree/2^j}{BlocksFree}$$

where $TotalFree$ is the number of free pages, j is the order of the desired allocation and $BlocksFree$ is the number of contiguous regions stored on freelists. When $BlocksFree$ is 0, we define $F_i(j)$ to be 0. A negative value of $F_i(j)$ implies that the allocation can be satisfied and the fragmentation index is only meaningful when an allocation fails. A value tending towards 0 implies the allocation failed due to a lack of memory. A value tending towards 1 implies that the failure is due to fragmentation.

[2]Discussions on fragmentation are typically concerned with internal fragmentation where the percentage represents wasted memory. A percentage value for external fragmentation is not as meaningful because it depends on the request size.

Obviously the fewer times the F_i are calculated, the better.

4 Allocator Placement Policies

It is common for allocators to exploit known characteristics of the request stream to improve their efficiency. For example, allocation size and the relative time of the allocation have been used to heuristically group objects of an expected lifetime together [1]. Similar heuristics cannot be used within an operating system as it does not have the same distinctive phases as application programs have. There is also little correlation between the size of an allocation and its expected use. However, operating system allocations do have unique characteristics that may be exploited to control placement thereby reducing fragmentation.

First, certain pages can be freed on demand; saved to backing storage; or discarded. Second, a large amount of kernel allocations are for caches, such as the buffer and inode caches which may be reclaimed on demand. Since it is known in advance what the page will be used for, an anti-fragmentation strategy can group pages by *allocation type*. We define three types of reclaimability

Easy to reclaim (EasyRclm) pages are allocated directly for a user process. Almost all pages mapped to a userspace page table and disk buffers, but not their management structures, are in this category.

Kernel reclaimable (KernRclm) pages are allocated for the kernel but can often be reclaimed on demand. Examples include inodes, buffer head and directory entry caches. Other examples, not applicable to Linux, include kernel data and PTEs where the system is capable of paging them to swap.

Kernel non-reclaimable (KernNoRclm) pages are essentially impossible to reclaim on demand.

To distinguish among the reclamation types, additional GFP flags are used when calling `alloc_pages()`. For simplicity, the strategies presented here treat KernNoRclm and KernRclm the same so we use only one flag `GFP_EASYRCLM` to distinguish between user and kernel allocations. Variations exist that deal with all three reclamation types, but the resulting code is relatively more complex.

Allocation requests that specify the `GFP_EASYRCLM` flag include requests for buffer pages, process faulted pages, high pages allocated with `alloc_zeroed_user_highpage` and shared memory pages. The strategies principally differ in the semantics of the GFP flag and its treatment in the implementation.

5 Anti-Fragmentation With Lists

The binary buddy allocator maintains *max_order* lists of free blocks of each power-of-two from 2^0 to $2^{max\_order-1}$. Instead of one list at each order, this strategy uses two lists by extending `struct free_area`. At each order, one list is used to satisfy EasyRclm allocations and the second list is used for all other allocations. `struct per_cpu_pages` is similarly extended to have one list for EasyRclm and one for kernel allocations.

The difference in design between the standard and list-based anti-fragmentation allocator is illustrated in Figure 1. Where possible, allocations of a specified type use their own freelist but can steal pages from each other in low memory conditions. When allocated, `SetPageEasyRclm()` is called for EasyRclm allocations so that they will be freed back to the correct lists. The two lists mean that a page's buddy is likely to be of the same reclaimability. The success of this strategy depends on there being a large enough number of EasyRclm pages and that there are no prolonged bursts of requests for kernel pages leading to excessive stealing.

One advantage of this strategy is that a high order kernel allocation can push out EasyRclm pages to satisfy the allocation. The assumption is that high-order allocations during the lifetime of the system are short-lived. Performance regressions tests did not show any problems despite the allocator hot paths being affected by this strategy.

A disadvantage is related to the advantage. As kernel order-0 allocations can use the EasyRclm freelists, the strategy can break down if there are prolonged periods of small allocations without frees. The likelihood is also that long-term light loads, such as desktops running for a number of days will allow kernel pages to slowly leak to all areas of physical memory. Over time, the list-based strategy would have similar success rates to the standard allocator.

6 Anti-Fragmentation With Zones

The Linux kernel splits available memory into one or more *zones*, each representing memory with different usage limitations as shown in Figure 2. On a typical x86, we have `ZONE_DMA` representing memory capable of use for Direct Memory Access (DMA), `ZONE_NORMAL` representing memory which is directly accessible by the kernel, and `ZONE_HIGHMEM` covering the remainder. Each zone has its own set of lists for the buddy allocator to track free memory within the zone.

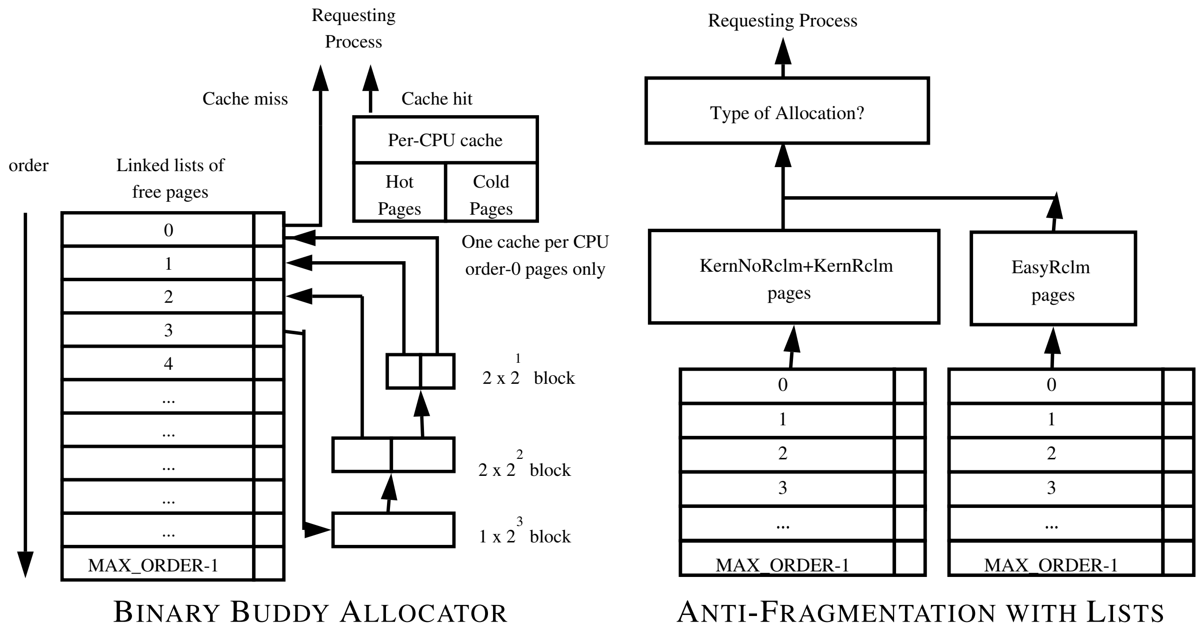


Figure 1: Comparison of the standard and list-based anti-frag allocators

This strategy introduces a new memory zone, ZONE_EASYRCLM, to contain EasyRclm pages as illustrated in Figure 3. EasyRclm allocations that cannot be satisfied from this zone fallback to regular zones, but non-EasyRclm allocations cannot use ZONE_EASYRCLM. This is a crucial difference between the list-based and zone-based strategies for anti-fragmentation as list-based allows stealing in both directions.

While booting, the system memory is split into portions required by the kernel for its operation and that which will be used for EasyRclm allocations. The size of the kernel portion is defined by the system administrator via the kernelcore= kernel parameter, which bounds the memory placed in the standard zones; the remaining memory constitutes ZONE_EASYRCLM. If kernelcore= is not specified, no pages are placed in ZONE_EASYRCLM.

The principal advantage of this strategy are that it provides a high likelihood of being able to reclaim appropriately sized portions of ZONE_EASYRCLM for any higher order allocation if the high-order allocation is also easily reclaimable. Another significant advantage is that ZONE_EASYRCLM may be used for HugeTLB page allocations as they do not worsen the fragmentation state of the system in a meaningful way. This allows us to use the ZONE_EASYRCLM as a “soft allocation” zone from the HugeTLB pool to expand into.

One disadvantage is similar to the HugeTLB pool sizing problem because the usage of the system must be known in advance. Sizing is workload dependant and performance may suffer if an inappropriate size is specified with kernelcore=. The second major disadvantage is that the strategy does not provide any help for high-order kernel allocations.

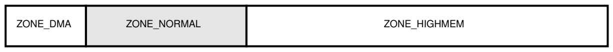


Figure 2: Standard Linux kernel zone layout

CPU	Xeon® 2.8GHz
# Physical CPUs	2
# CPUs	4
Main Memory	1518MiB

X86-BASED TEST MACHINE

CPU	Power5® PPC64 1.9GHz
# Physical CPUs	2
# CPUs	4
Main Memory	4019MiB

POWER5-BASED TEST MACHINE

Figure 4: Specification of Test Machines

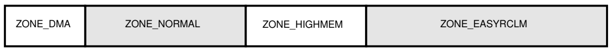


Figure 3: Easy Reclaim zone layout

7 Experimental Methodology

The strategies were evaluated using five tests, two related to performance and three related to the system's ability to satisfy large contiguous allocations. The system is cleanly booted at the beginning of a single set of tests. Each of the five tests are run in order without intervening reboots to maximise the chances of the system suffering fragmentation. The tests are as follows

kbuild is similar to kernbench and it measures the time taken to extract and build a kernel. The test gives an overall view of the performance of a kernel, including the rate the kernel is able to satisfy allocations.

AIM9 is a micro-benchmark that includes tests for VM-related operations like page allocation and the time taken to call `brk()`. AIM9 is a good barometer for performance regressions. Crucially, it is sensitive to regressions in the page allocator paths.

HugeTLB-Capability is a kernel compile based benchmark. For every 250MiB of physical memory, a kernel compile is executed (in parallel, simultaneously). During the compile, one attempt is made to grow the HugeTLB page pool from 0 by echoing a large number to `/proc/sys/vm/nr_hugepages`. After the re-size attempt, the pool is shrunk back to 0. The kernel compiles are then stopped and an attempt is made to grow the pool while the system is under no significant load. A zero-filled file that is the same size as physical memory is then created with `dd`, then deleted, before a third attempt is made to re-size the HugeTLB pool. This test determines how capable the system is of allocating HugeTLB pages at run-time using the conventional interfaces.

Highalloc-Stress is a kernel compile based benchmark. Kernel compiles are started as in the HugeTLB-Capability test, plus `updatedb` is also run in the background. A kernel module is loaded to aggressively allocate as many HugeTLB-sized pages as the system has by calling `alloc_pages()`. These persistent attempts force `kswapd` to start reclaiming as well as triggering direct reclaim which does not occur when resizing the HugeTLB pool via `/proc/sys/vm/nr_hugepages`. $F_u(hugetlb\_order)$ is calculated at each allocation attempt and $F_i(hugetlb\_order)$ is calculated at each failure (see Section 3). The results are graphed at the end of the test. This test indicates how many HugeTLB pages could be allocated under the best of circumstances.

HotRemove-Capability is a memory hotplug remove test. For each section of memory reported in `/sys/devices/system/memory`, an attempt is made to off-line the memory. Assuming the kernel supports hotplug-remove, a report states how many sections and what percentage of memory was off-lined. The base kernel used for this paper was `2.6.16-rc6` which did not support hotplug remove, so no results were produced and it will not be discussed further.

All of these benchmarks were run using driver scripts from VMRegress 0.36[3] in conjunction with the same system that generates the reports on `http://test.kernel.org`. Two machines were used to run the benchmarks based on the x86 and Power5® architectures as detailed in Figure 4. In both cases, the tests were run and results collected with scripts to minimise variation and prevent bias during testing. Four sets of configurations were run on each architecture

1. List-based strategy under light load
2. List-based strategy under heavy load
3. Zone-based with no `kernelcore` specified giving a `ZONE_EASYRCLM` with zero pages.
4. Zone-based with `kernelcore=1024MB` on x86 and `kernelcore=2048MB` on PPC64.

The list-based strategy is tested under light and heavy loads to determine if the strategy breaks down under pressure. We anticipated the results of the benchmarks to be similar if no breakdown was occurring. The zone-based strategy is tested with and without `kernelcore` to show that `ZONE_EASYRCLM` is behaving as expected and that the existence of the zone does not incur a performance penalty. The choice of 2048MB on PPC64 is 50% of physical memory. The choice of 1024MB on x86 is to give some memory to `ZONE_EASYRCLM`, but to leave some memory in `ZONE_HIGHMEM` for PTE use as `CONFIG_HIGHPTE` was set.

[3]http://www.csn.ul.ie/~mel/projects/vmregress/vmregress-0.37.tar.gz

8 Results

On the successful completion of a test run, a summarised report is generated similar[4] to the one shown in Figure 11. These reports get aggregated into the graphs shown in Figures 12 and 13. For each architecture the graphs show how the two strategies compare against the base allocator in terms of performance and the ability to satisfy HugeTLB allocations. These graphs will be the focus of our discussion on performance in Section 8.1.

Figures 5 and 6 shows the values of $F_u(hugetlb\_order)$ at each allocation attempt during the Highalloc-Stress Test while the system was under no load. Note that in all cases, the starting value of $F_u(hugetlb\_order)$ is close to 1 indicating that free memory was not in large contiguous regions after the kernel compiles were stopped. The value drops over time as pages are reclaimed and buddies coalesce. Kernels using anti-fragmentation strategies had a higher rate of decline for the value of $F_u(hugetlb\_order)$, which implies that the anti-fragmentation strategies had a measure of success. These figures will be the focus of our discussion on the ability of the system to satisfy requests for contiguous regions in Section 8.2.

Finally, Figures 7 and 8 show the value of $F_i(hugetlb\_order)$ at each allocation failure during the Highalloc-Stress Test while the system was under no load. These illustrate the root cause of the allocation failures and are discussed in Section 8.3.

8.1 Performance

On both architectures, absolute performance was comparable. The "KBuild Comparison"

[4]Edited to fit

graphs in Figures 12 and 13 show the timings were within seconds of each other and this was consistent among runs. The "AIM9 Comparison" graphs show that any regression was within 3% of the base kernel's performance. This is expected as that test varies by a few percent in each run and the results represent one run, not an average. This leads us to conclude that neither list-based nor zone-based has a significant performance penalty on either x86 or PPC64 architectures, at least for our sample workloads.

8.2 Free Space Usability

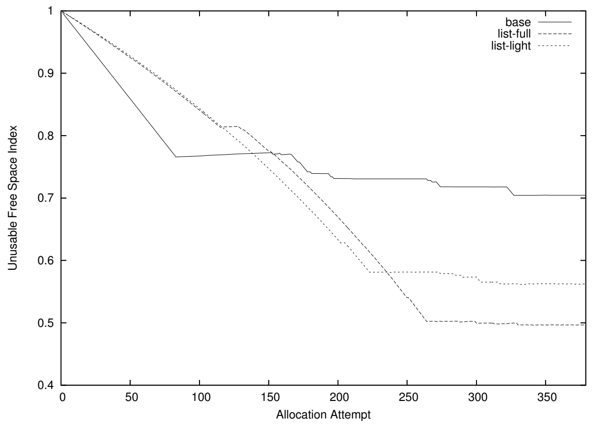
In general, zone-based was more predictable and reliable at providing contiguous free space. On both architectures, the zone-based anti-fragmentation kernels were able to allocate almost all of the pages in `ZONE_EASYRCLM` at rest after the tests. As shown on Figure 6, 0.66 was the final value of $F_u(hugetlb\_order)$ on PPC64 with half of physical memory in `ZONE_EASYRCLM`. We would expect it to reach 0.50 after multiple HugeTLB allocation attempts. Without specifying `kernelcore`, the scheme made no difference to absolute performance or fragmentation as `ZONE_EASYRCLM` is empty.

The list-based strategy was potentially able to reduce fragmentation throughout physical memory. On x86, list-based anti-fragmentation kept overall fragmentation lower than zone-based but it was only fractionally better on the PPC64 than the standard allocator. An examination of the x86 "High Allocation Stress Test Comparison Test" report in Figure 12 hints why. On x86, advantage is being taken of the existing zone-based groupings of allocation types in Normal and HighMem. Effectively, it was using a simple zone-based anti-fragmentation that did not take PTEs into account. The list-based strategy succeeds on x86 because it keeps the PTE pages in HighMem grouped together in addition to some success in `ZONE_NORMAL`. Nevertheless, the strategy clearly breaks down in `ZONE_NORMAL` due to large amounts of kernel allocations falling back to the EasyRclm freelists in low-memory situations. The breakdown is is illustrated by the different values of $F_u(hugetlb\_order)$ after the different loads where similar values would be expected if no breakdown was occurring. Figure 5 shows that the light-load performed *worse* than full-load due to the unpredictability of the strategy. On an earlier run, the list-based strategy under light load was able to allocate 119 HugeTLB pages from `ZONE_NORMAL` but only 77 after full-load.

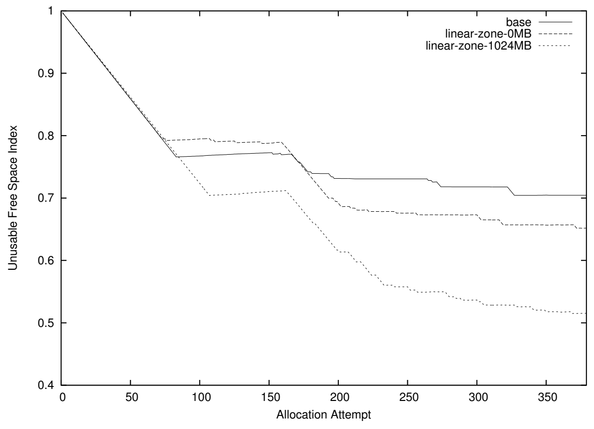
Under load, neither scheme was significantly better than the other at keeping free areas contiguous. This is because we were depending on the LRU-approximation to reclaim a contiguous region. Under load, zone-based was generally better because page reclaim was able to reclaim within `ZONE_EASYRCLM` but the list-based strategy did not have the same focus. With either anti-fragmentation strategy, LRU simply is not suitable for reclaiming contiguous regions and an alternative strategy is discussed in Section 10.

8.3 Fragmentation Index at Failure

Figures 7 and 8 clearly show that allocations failed with both strategies due to fragmentation and not lack of memory. By design, the zone-based strategy does not reduce fragmentation in the kernel zones. When an allocation fails at rest, it is because `ZONE_EASYRCLM` is likely nearly depleted and we are looking at the high fragmentation in the kernel zones. The figures for list-based implied that, under load, fragmentation had crept into all zones which means the strategy broke down due to excessive stealing.

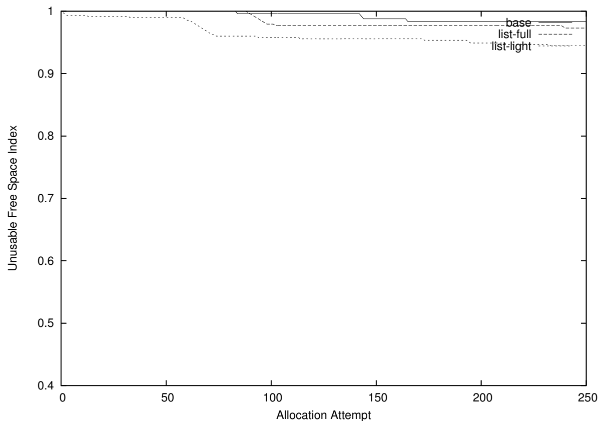


X86 LIST-BASED

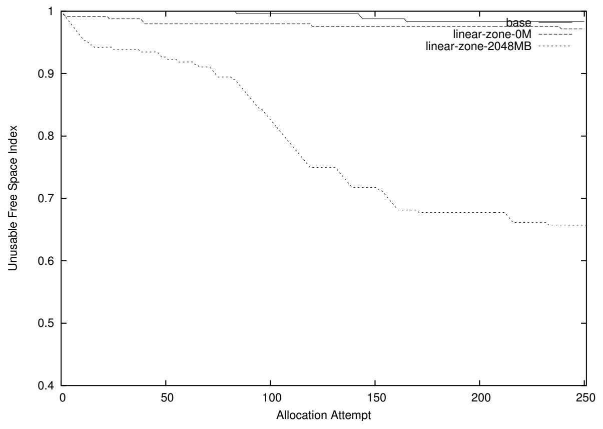


X86 ZONE-BASED

Figure 5: x86 Unusable Free Space Index During Highalloc-Stress Test, System At Rest

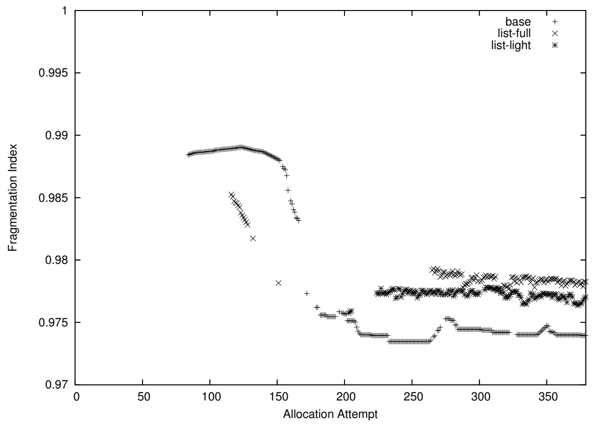


PPC64 LIST-BASED

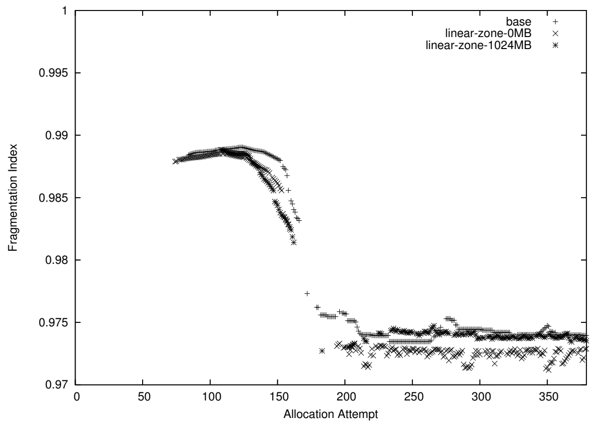


PPC64 ZONE-BASED

Figure 6: PPC64 Unusable Free Space Index During Highalloc-Stress Test, System At Rest

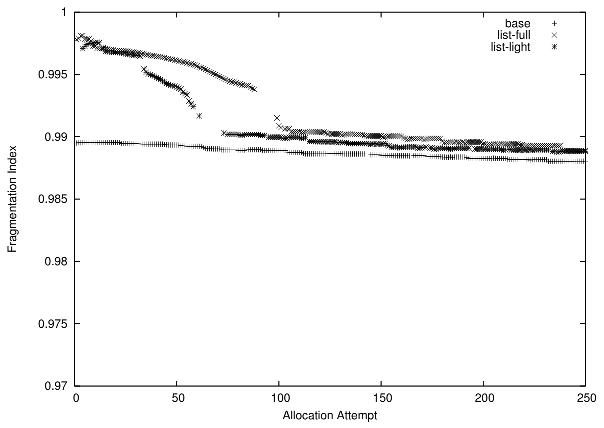


X86 LIST-BASED

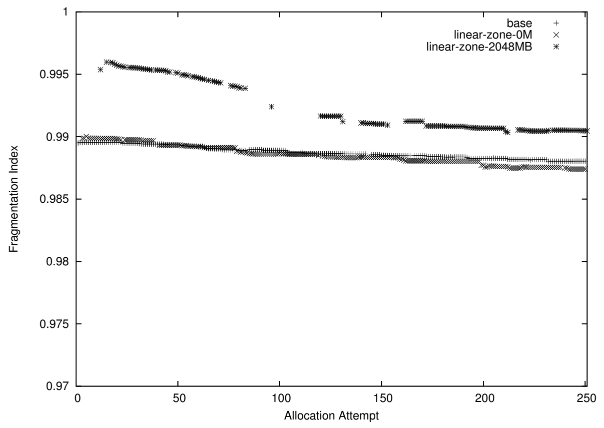


X86 ZONE-BASED

Figure 7: x86 Fragmentation Index at Allocation Failures During Highalloc-Stress Test



PPC64 LIST-BASED



PPC64 ZONE-BASED

Figure 8: PPC64 Fragmentation Index at Allocation Failures During Highalloc-Stress

9 Results Conclusions

The two strategies had different advantages and disadvantages but both were able to increase availability of HugeTLB pages. The fact that list-based does not require configuration and works on all of memory makes it desirable but our figures show that it breaks down in its current implementation. Once correctly configured, zone-based is more reliable even though it does not help high-order kernel allocations.

The zone-based strategy is currently the best available solution. In the short-to-medium term, the zone-based strategy creates a soft-area that can satisfy HugeTLB allocations on demand. In the long-term, we intend to develop a strategy that takes the best from both approaches without incurring a performance regression.

10 Linear Reclaim

Anti-fragmentation improves our chances of finding contiguous regions of memory that may be reclaimed to satisfy a high order allocation. However, the existing LRU-approximation algorithm for page reclamation is not suitable for finding contiguous regions.

In the *worst-case* scenario, the LRU list contains randomly ordered pages across the system so the release of pages will also be in random order. To free a contiguous region of 2^j pages within a zone containing N pages, we may need to release $F_r(j)$ pages in that zone where

$$F_r(j) = (\frac{N}{2^j} * (2^j - 1)) + 1$$

The table in Figure 9 shows the relative proportion of memory we will need to reclaim before we can guarantee to free a contiguous region of sufficient size for the specified order. We can see that beyond the lowest orders we need to reclaim most pages in the system to guarantee freeing pages of the desired order. Order 10 and 12 are interesting as they represent the HugeTLB page sizes for x86 and PPC64 respectively. The average case is not this severe, but a detailed analysis of the average case is beyond the scope of this paper.

We introduced an alternative reclaim algorithm called *Linear Reclaim* designed to target larger

Order	Percentage
1	50.00
2	75.00
3	87.50
4	93.75
5	96.88
6	98.44
10	99.90
12	99.98

Figure 9: Reclaim Difficulty

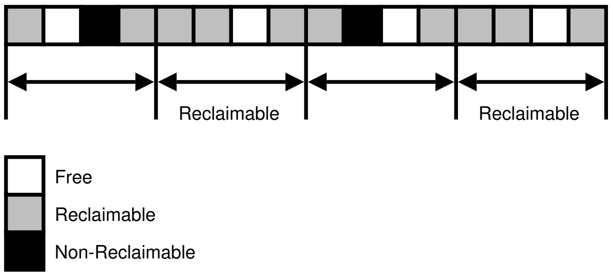


Figure 10: Linear Reclaim

contiguous regions of pages. It is used when the failing allocation is of order 3 or greater. With linear reclaim we view the entire memory space as a set of contiguous regions, each of the size we are trying to release. For each region, we check if all of the pages are likely to be reclaimable or are already free. If so, the allocated pages are removed from the LRU and an attempt is made to reclaim them. This continues until a proportion of the contiguous regions have been scanned.

In our example in Figure 10, linear reclaim will only attempt to reclaim pages in the second and fourth regions, applying reclaim to all the pages in the selected region at the same time. It is clear that in the case where reclaim succeeds we should be able to free the region by releasing just its pages which is significantly less than that required with LRU-based reclaim.

An early proof-of-concept implementation of linear reclaim was promising. A HugeTLB-capability test was run on the x86 machine. Under load, a clean kernel was able to allocate 6 HugeTLB pages, the zone-based anti-fragmentation allocator was able to allocate 10 HugeTLB pages and with both zone-based anti-fragmentation and linear-reclaim, it was able to allocate 41 HugeTLB pages. We do not have detailed timing information but early indications are that linear reclaim is able to satisfy allocation requests faster but spends more time scanning than the existing page reclamation policy before a failure. In summary, linear reclaim is promising, but needs further development.

11 Future Work

We intend to develop the zone-based anti-fragmentation strategy further. The patches that exist at the time of writing include some complex architecture-specific code that calculate the size of `ZONE_EASYRCLM`. As the code for sizing zones and memory holes in each architecture is similar, we are developing code to calculate the size of zones and holes in an architecture-independent fashion. Our initial patches show a net reduction of code.

Once an anti-fragmentation strategy is in place, we would like to develop the linear reclaim scanner further as LRU reclaims far too much memory to satisfy a request for a contiguous region. Our current testing strategy records how long it takes to satisfy a large allocation and we anticipate linear reclaim will show improvements in those figures.

In a perfect world, with everything in place, the plan is to work on the transparent support of HugeTLB pages in Linux. Although there are known applications that benefit from this such as database and java-based software, we would also like to show benefits for desktop software such as X.

We will then determine if there is a performance case for the use of higher-order allocations by the kernel. If there is, we will revisit the list-based approach and determine if a more general solution can be developed to control fragmentation throughout the system, and not just in pre-configured zones.

Acknowledgements

We would like to thank Nishanth Aravamudan for reviewing a number of drafts of this paper and his suggestions on how to improve the quality. We would like to thank Dave Hansen for his clarifications on the content, particularly on the size of SPARSEMEM memory sections on x86. Finally, we would like to thank Paul McKenney for his in-depth commentary on an earlier version of the paper and particularly for his feedback on Section 3.

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of IBM.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be the trademarks or service marks of others.

References

[1] D. A. Barrett and B. G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI*, pages 187–196, 1993.

[2] J. B. Chen, A. Borg, and N. P. Jouppi. A simulation based study of TLB performance. In *ISCA*, pages 114–123, 1992.

[3] D. G. Korn and K.-P. Bo. In search of a better malloc. In *Proceedings of the Summer 1985 USENIX Conference*, pages 489–506, 1985.

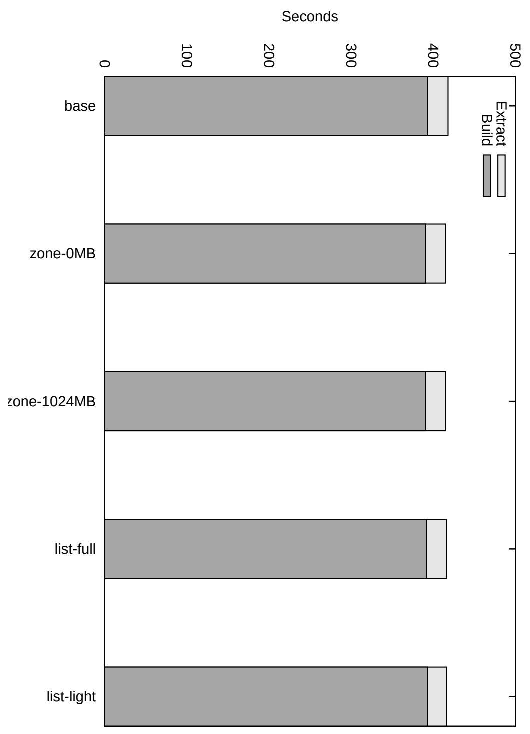
[4] M. K. McKusick. *The design and implementation of the 4.4BSD operating system.* Addison-Wesley, 1996.

[5] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, 1977.

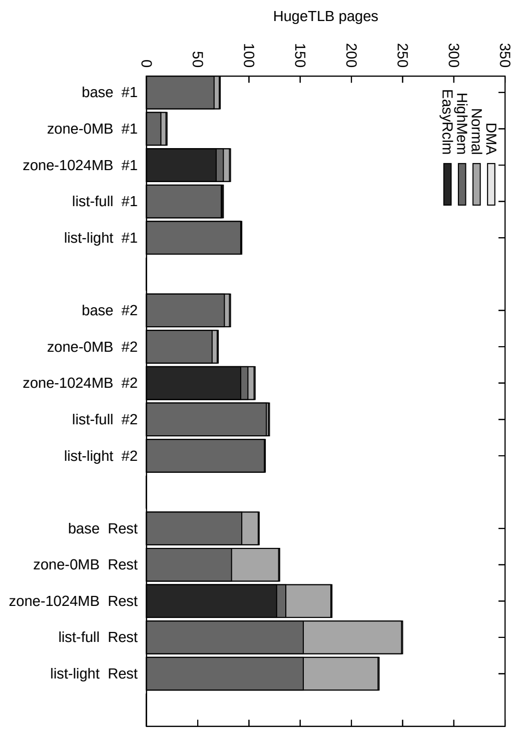
[6] B. Randell. A note on storage fragmentation and program segmentation. *Commun. ACM*, 12(7):365–369, 1969.

```
Kernel comparison report
------------------------
Architecture:            x86
Huge Page Size:          4 MB
Physical memory:         1554364 KB
Number huge pages:       379

KBuild Comparison
-----------------
                               2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Time taken to extract kernel:                25                   24                      24
Time taken to build kernel:                 393                  391                     391

AIM9 Comparison
---------------
                 2.6.16-rc6-clean    zone-0MB         zone-1024MB
 1 creat-clo           105965.67   105866.67 -0.09%    106500.00  0.50% File Creations and Closes/s
 2 page_test           259306.67   271558.07  4.72%    258300.28 -0.39% System Allocations & Pages/s
 3 brk_test           1666572.24  1866883.33 12.02%   1880766.67 12.85% System Memory Allocations/s
 4 jmp_test          14805650.00 13949966.67 -5.78%  15088700.00  1.91% Non-local gotos/second
 5 signal_test         286252.29   280183.33 -2.12%    282950.00 -1.15% Signal Traps/second
 6 exec_test              131.79      131.98  0.14%       131.68 -0.08% Program Loads/second
 7 fork_test             3857.69     3842.69 -0.39%      3862.69  0.13% Task Creations/second
 8 link_test            21291.90    21693.58  1.89%     21499.37  0.97% Link/Unlink Pairs/second

High Allocation Stress Test Comparison
--------------------------------------
HighAlloc Under Load Test Results Pass 1
                           2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Order                                    10                   10                      10
Success allocs                           72                   20                      82
Failed allocs                           307                  359                     297
DMA zone allocs                           1                    1                       1
Normal zone allocs                        5                    5                       6
HighMem zone allocs                      66                   14                       7
EasyRclm zone allocs                      0                    0                      68
% Success                                18                    5                      21
HighAlloc Under Load Test Results Pass 2
                           2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Order                                    10                   10                      10
Success allocs                           82                   70                     106
Failed allocs                           297                  309                     273
DMA zone allocs                           1                    1                       1
Normal zone allocs                        5                    5                       6
HighMem zone allocs                      76                   64                       7
EasyRclm zone allocs                      0                    0                      92
% Success                                21                   18                      27
HighAlloc Test Results while Rested
                           2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
Order                                    10                   10                      10
Success allocs                          110                  130                     181
Failed allocs                           269                  249                     198
DMA zone allocs                           1                    1                       1
Normal zone allocs                       16                   46                      44
HighMem zone allocs                      93                   83                       9
EasyRclm zone allocs                      0                    0                     127
% Success                                29                   34                      47

HugeTLB Page Capability Comparison
----------------------------------
                                  2.6.16-rc6-clean  2.6.16-rc6-zone-0MB  2.6.16-rc6-zone-1024MB
During compile:                                  5                    5                       5
At rest before dd of large file:                51                   52                      48
At rest after  dd of large file:                67                   64                      92
```

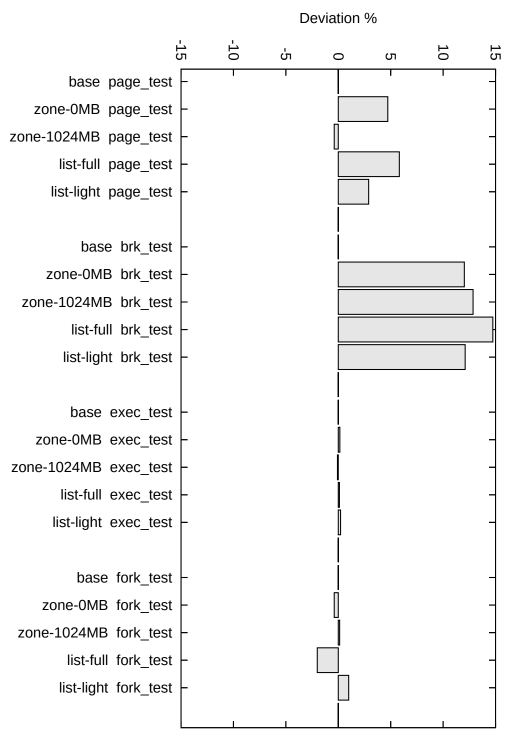
Figure 11: Example Kernel Comparison Report



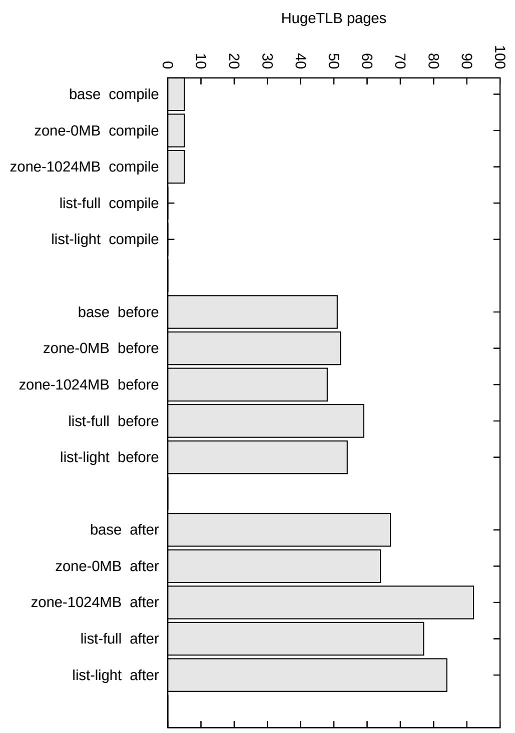
KBUILD COMPARISON



HIGH ALLOCATION STRESS TEST COMPARISON

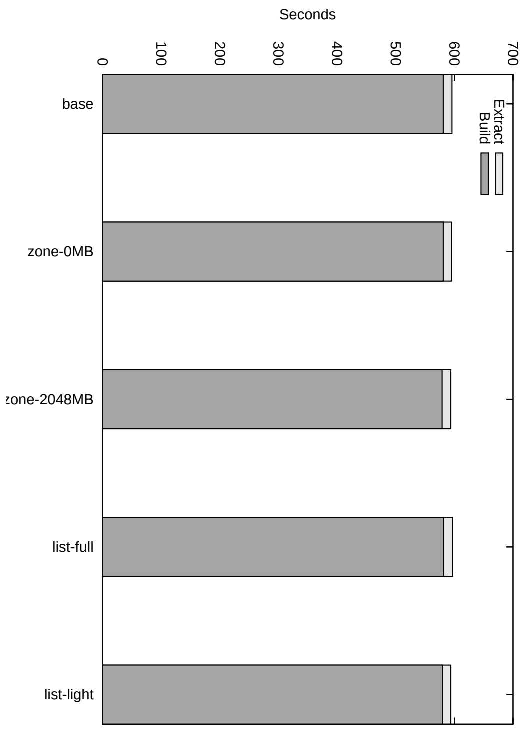


AIM9 COMPARISON

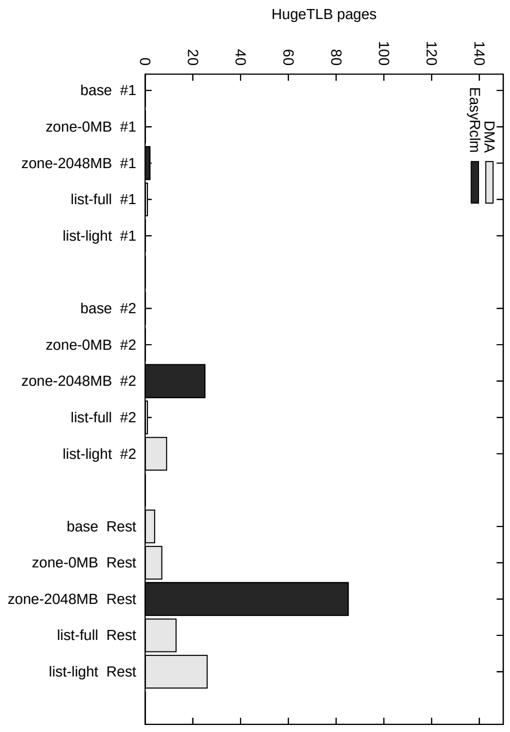


HUGETLB PAGE CAPABILITY COMPARISON

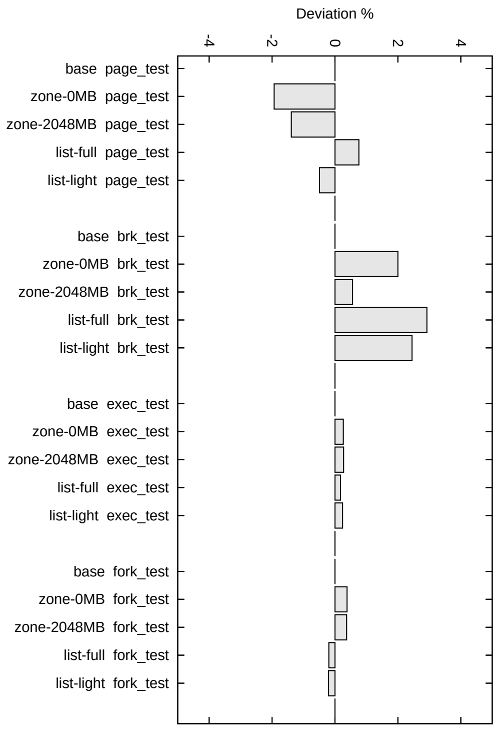
Figure 12: Anti-Fragmentation Strategy Comparison on x86



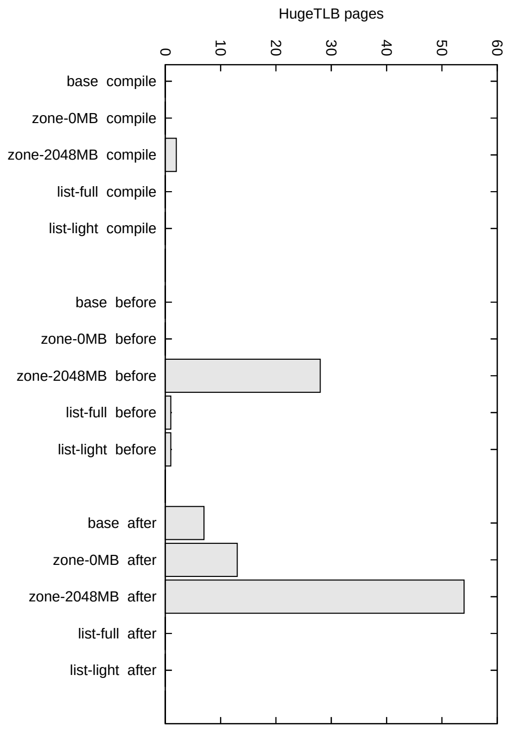
KBUILD COMPARISON



HIGH ALLOCATION STRESS TEST COMPARISON



AIM9 COMPARISON



HUGETLB PAGE CAPABILITY COMPARISON

Figure 13: Anti-Fragmentation Strategy on PPC64

GIT—A Stupid Content Tracker

Junio C. Hamano
Twin Sun, Inc.
junio@twinsun.com

Abstract

Git was hurriedly hacked together by Linus Torvalds, after the Linux kernel project lost its license to use BitKeeper as its source code management system (SCM). It has since quickly grown to become capable of managing the Linux kernel project source code. Other projects have started to replace their existing SCMs with it.

Among interesting things that it does are:

1. giving a quick whole-tree diff,
2. quick, simple, stupid-but-safe merge,
3. facilitating e-mail based patch exchange workflow, and
4. helping to pin-point the change that caused a particular bug by a bisection search in the development history.

The core git functionality is implemented as a set of programs to allow higher-layer systems (Porcelains) to be built on top of it. Several Porcelains have been built on top of git, to support different workflows and individual taste of users. The primary advantage of this architecture is ease of customization, while keeping the repositories managed by different Porcelains compatible with each other.

The paper gives an overview of how git evolved and discusses the strengths and weaknesses of its design.

1 Low level design

Git is a "stupid content tracker." It is designed to record and compare the whole tree states efficiently. Unlike traditional source code control systems, its data structures are not geared toward recording changes between revisions, but for making it efficient to retrieve the state of individual revisions.

The unit in git storage is an *object*. It records:

- `blob` – the contents of a file (either the contents of a regular file, or the path pointed at by a symbolic link).
- `tree` – the contents of a directory, by recording the mapping from names to objects (either a blob object or a tree object that represents a subdirectory).
- `commit` – A commit associates a tree with meta-information that describes how the tree came into existence. It records:
  - The tree object that describes the project state.

    - The author name and time of creation of the content.
    - The committer name and time of creation of the commit.
    - The parent commits of this commit.
    - The commit log that describes why the project state needs to be changed to the tree contained in this commit from the trees contained in the parent commits.
- `tag` – a tag object names another object and associates arbitrary information with it. A person creating the tag can attest that it points at an authentic object by GPG-signing the tag.

Each object is referred to by taking the SHA1 hash (160 bits) of its internal representation, and the value of this hash is called its object name. A tree object maps a pathname to the object name of the blob (or another tree, for a subdirectory).

By naming a single tree object that represents the top-level directory of a project, the entire directory structure and the contents of any project state can be recreated. In that sense, a tree object is roughly equivalent to a tarball.

A commit object, by tying its tree object with other commit objects in the ancestry chain, gives the specific project state a point in project history. A merge commit ties two or more lines of developments together by recording which commits are its parents. A tag object is used to attach a label to a specific commit object (e.g. a particular release).

These objects are enough to record the project history. A project can have more than one lines of developments, and they are called *branches*. The latest commit in each line of development is called the head of the branch, and a repository keeps track of the heads of currently active branches by recording the commit object names of them.

To keep track of what is being worked on in the user's working tree, another data structure called *index*, is used. It associates pathnames with object names, and is used as the staging area for building the next tree to be committed.

When preparing an index to build the next tree, it also records the `stat(2)` information from the working tree files to optimize common operations. For example, when listing the set of files that are different between the index and the working tree, git does not have to inspect the contents of files whose cached `stat(2)` information match the current working tree.

The core git system consists of many relatively low-level commands (often called Plumbing) and a set of higher level scripts (often called Porcelain) that use Plumbing commands. Each Plumbing command is designed to do a specific task and only that task. The Plumbing commands to move information between the recorded history and the working tree are:

- Files to index. `git-update-index` records the contents of the working tree files to the index; to write out the blob recorded in the index to the working tree files, `git-checkout-index` is used; and `git-diff-files` compares what is recorded in the index and the working tree files. With these, an index is built that records a set of files in the desired state to be committed next.
- Index to recorded history. To write out the contents of the index as a tree object, `git-write-index` is used; `git-commit-tree` takes a tree object, zero or more commit objects as its parents, and the commit log message, and creates a commit object. `git-diff-index`

compares what is recorded in a tree object and the index, to serve as a preview of what is going to be committed.

- Recorded history to index. The directory structure recorded in a tree object is read by `git-read-tree` into the index.
- Index to files. `git-checkout-index` writes the blobs recorded in the index to the working tree files.

By tying these low-level commands together, Porcelain commands give usability to the whole system for the end users. For example, `git-commit` provides a UI to ask for the commit log message, create a new commit object (using `git-write-tree` and `git-commit-tree`), and record the object name of that commit as the updated topmost commit in the current line of development.

2 Design Goals

From the beginning, git was designed specifically to support the workflow of the Linux kernel project, and its design was heavily influenced by the common types of operations in the kernel project. The statistics quoted below are for the 10-month period between the beginning of May 2005 and the end of February 2006.

- The source tree is fairly large. It has approximately 19,000 files spread across 1,100 directories, and it is growing.
- The project is very active. Approximately 20,000 changes were made during the 10-month period. At the end of the examined period, around 75% of the lines are from the version from the beginning of the period, and the rest are additions and modifications.
- The development process is highly distributed. The development history led to v2.6.16 since v2.6.12-rc2 contains changes by more than 1,800 authors that were committed by a few dozen people.
- The workflow involves many patch exchanges through the mailing list. Among 20,000 changes, 16,000 were committed by somebody other than the original author of the change.
- Each change tends to touch only a handful files. The source tree is highly modular and a change is often very contained to a small part of the tree. A change touches only three files on average, and modifies about 160 lines.
- The tree reorganization by addition and deletion is not so uncommon, but often happens over time, not as a single rename with some modifications. 4,500 files were added or deleted, but less than 600 were renamed.
- The workflow involves frequent merges between subsystem trees and the mainline. About 1,500 changes are merges (7%).
- A merge tends to be straightforward. The median number of paths involved in the 1,500 merges was 185, and among them, only 10 required manual inspection of content-level merges.

Initial design guidelines came from the above project characteristics.

- A few dozen people playing the integrator role have to handle work by 2,000 contributors, and it is paramount to make it efficient form them to perform common operations, such as patch acceptance and merging.

- Although the entire project is large, individual changes tend to be localized. Supporting patch application and merging with a working tree with local modifications, as long as such local modifications do not interfere with the change being processed, makes the integrators' job more efficient.

- The application of an e-mailed patch must be very fast. It is not uncommon to feed more than 1,000 changes at once during a sync from the `-mm` tree to the mainline.

- When existing contents are moved around in the project tree, renaming of an entire file (with or without modification at the same time) is not a majority. Other content movements happen more often, such as consolidating parts of multiple files to one new file or splitting an existing files into multiple new files. Recording file renames and treating them specially does not help much.

- Although the merge plays an important role in building the history of the project, clever merge algorithms do not make much practical difference, because majority of the merges are trivial; nontrivial cases need to be examined carefully by humans anyway, and the maintainer can always respond, "This does not apply, please rework it based on the latest version and resubmit." Faster merge is more important, as long as it does not silently merge things incorrectly.

- Frequent and repeated merges are the norm. It is important to record what has already been merged in order to avoid having to resolve the same merge conflicts over and over again.

- Two revisions close together tend to have many common directories unchanged between them. Tree comparison can take advantage of this to avoid descending into subdirectories that are represented by the same tree object while examining changes.

3 Evolution

The very initial version of git, released by Linus Torvalds on April 7, 2005, had only a handful of commands to:

- initialize the repository;
- update the index to prepare for the next tree;
- create a tree object out of the current index contents;
- create a commit object that points at its tree object and its parent commits;
- print the contents of an object, given its object name;
- read a tree object into the current index;
- show the difference between the index and the working tree.

Even with this only limited set of commands, it was capable of hosting itself. It needed scripting around it even for "power users."

By the end of the second week, the Plumbing level already had many of the fundamental data structure of today's git, and the initial commit of the modern Linux kernel history hosted on git (v2.6.12-rc2) was created with this version. It had commands to:

- read more than one tree object to process a merge in index;

- perform content-level merges by iterating over an unmerged index;
- list the commit ancestry and find the most recent common commit between two lines of development;
- show differences between two tree objects, in the raw format;
- fetch from a remote repository over rsync and merge the results.

When two lines of development meet, git uses the index to match corresponding files from the common ancestor (merge base) and the tips of the two branches. If one side changed a file while the other side didn't, which often happens in a big project, the merge algorithm can take the updated version without looking at the contents of the file itself. The only case that needs the content-level merge is when both side changed the same file. This tree merge optimization is one of the foundations of today's git, and it was already present there. The first ever true git merge in the Linux kernel repository was made with this version on April 17, 2005.

By mid May, 2005, it had commands to:

- fetch objects from remote repositories over HTTP;
- create tags that point at other objects;
- show differences between the index and a tree, working tree files and a tree, in addition to the original two tree comparison commands—both raw and patch format output were supported;
- show the commit ancestry along with the list of changed paths.

By the time Linus handed the project over to the current maintainer in late July 2005, the core part was more or less complete. Added during this period were:

- packed archive for efficient storage, access, and transfer;
- the git "native" transfer protocol, and the `git-daemon` server;
- exporting commits into the patch format for easier e-mail submission.
- application of e-mailed patches.
- rename detection by diff commands.
- more "user friendliness" layer commands, such as `git-add` and `git-diff` wrappers.

The evolution of git up to this point primarily concentrated on supporting the people in the integrator role better. Support for individual developers who feed patches to integrators was there, but providing developers with more pleasant user experiences was left to third-party Porcelains, most notably Cogito and StGIT.

4 Features and Strengths

This section discusses a few examples of how the implementation achieves the design goals stated earlier.

4.1 Patch flows

There are two things git does to help developers with the patch-based workflow.

Generating a patch out of a git-managed history is done by using the `git-diff-tree` command, which knows how to look at only subtrees that are actually different in two trees for efficient patch generation.

The diff commands in git can optionally be told to detect file renames. When a file is renamed and modified at the same time, with this option, the change is expressed as a diff between the file under the old name in the original tree and the file under the new name in the updated tree. Here is an example taken from the kernel project:

```
5e7b83ffc67e15791d9bf8b2a18e4f5fd0eb69b8
diff --git a/arch/um/kernel/sys_call_table....
similarity index 99%
rename from arch/um/kernel/sys_call_table.c
rename to arch/um/sys-x86_64/sys_call_table.c
index b671a31..3f5efbf 100644
--- a/arch/um/kernel/sys_call_table.c
+++ b/arch/um/sys-x86_64/sys_call_table.c
@@ -16,2 +16,8 @@ #include "kern_util.h"

+#ifdef CONFIG_NFSD
+#define NFSSERVCTL sys_nfsservctl
+#else
+#define NFSSERVCTL sys_ni_syscall
+#endif
+
 #define LAST_GENERIC_SYSCALL __NR_keyctl
```

This is done to help reviewing such a change by making it easier than expressing it as a deletion and a creation of two unrelated files.

The committer (typically the subsystem maintainer) keeps the tip of the development branch checked out, and applies e-mailed patches to it with `git-apply` command. The command checks to make sure the patch applies cleanly to the working tree, paths affected by the patch in the working tree are unmodified, and the index does not have modification from the tip of the branch. These checks ensure that after applying the patch to the working tree and the index, the index is ready to be committed, even when there are unrelated changes in the working tree. This allows the subsystem maintainer to be in the middle of doing his own work and still accept patches from outside.

Because the workflow git supports should not require all participants to use git, it understands both patches generated by git and traditional diff in unified format. It does not matter how the change was prepared, and does not negatively affect contributors who manage their own patches using other tools, such as Andrew Morton's patch-scripts, or quilt.

4.2 Frequent merges

A highly distributed development process involves frequent merges between different branches. Git uses its commit ancestry relation to find common ancestors of the branches being merged, and uses a three-way merge algorithm at two levels to resolve them. Because merges tend to happen often, and the subprojects are highly modular, most of the merges tend to deal with cases where only one branch modifies paths that are left intact by the other branch. This common case is resolved within the index, without even having to look at the contents of files. Only paths that need content-level merges are given to an external three-way merge program (e.g. "merge" from the RCS suite) to be processed. Similarly to the patch-application process, the merge can be done as long as the index does not have modification from the tip of the branch and there is no change to the working tree files that are involved in the merge, to allow the integrator to have local changes in the working tree.

While merging, if one branch renamed a file from the common ancestor while the other branch kept it at the same location (or renamed it to a different location), the merge algorithm notices the rename between the common ancestor and the tips of the branches, and applies three-way merge algorithm to merge the renames (i.e. if one branch renamed but the other kept it the same, the file is renamed). The experiences by users of this "merging renamed paths" feature is mixed. When merges are frequently done, it is more likely that the differences in the contents between the common ancestor and the tip of the branch is small enough that automated rename detector notices it.

When two branches are merged frequently with each other, there can be more than one closest common ancestor, and depending on which ancestor is picked, the three-way merge is known to produce different results. The merge algorithms git uses notice this case and try to be safe. The faster "resolve" algorithm leaves the resolution to the end-user, while the more careful "recursive" algorithm first attempts to merge the common ancestors (recursively—hence its name) and then uses the result as the merge base of the three-way merge.

4.3 Following changes

The project history is represented as a parent-child ancestry relation of commit objects, and the Plumbing command `git-rev-list` is used to traverse it. Because detecting subdirectory changes between two trees is a very cheap operation in git, it can be told to ignore commits that do not touch certain parts of the directory hierarchy by giving it optional pathnames. This allows the higher-level Porcelain commands to efficiently inspect only "interesting" commits more closely.

The traversal of the commit ancestry graph is also done while finding a regression, and is used by the `git-bisect` command. This traversal can also be told to omit commits that do not touch a particular area of the project directory; this speeds up the bug-hunting process when the source of the regression is known to be in a particular area.

4.4 Interoperating with other SCM

A working tree checked out from a foreign SCM system can be made into a git repository. This allows an individual participant of a project whose primary SCM system is not git to manage his own changes with git. Typically this is done by using two git branches per the upstream branch, one to track the foreign SCM's progress, another to hold his own changes based on that. When changes are ready, they are fed back by the project's preferred means, be it committing into the foreign SCM system or sending out a series of patches via e-mail, without affecting the workflow of the other participants of the projects. This allows not just distributed development but distributed choice of SCM. In addition, there are commands to import commits from other SCM systems (as of this writing, supported systems are: GNU arch, CVS, and Subversion), which helps to make this process smoother.

There also is an emulator that makes a git repository appear as if it is a CVS repository to remote CVS clients that come over the `pserver` protocol. This allows people more familiar with CVS to keep using it while others work in the same project that is hosted on git.

4.5 Interoperating among git users

Due to the clear separation of the Plumbing and Porcelain layers, it is easy to implement higher-level commands to support different workflows on top of the core git Plumbing commands. The Porcelain layer that comes with the core git requires the user to be fairly familiar with how the tools work internally, especially how the index is used. In order to make effective use of the tool, the users need to be aware that there are three levels of entities: the histories recorded in commits, the index, and the working tree files.

An alternative Porcelain, Cogito, takes a different approach by hiding the existence of the index from the users, to give them a more traditional two-level world model: recorded histories and the working tree files. This may fall down at times, especially for people playing the integrator role during merges, but gives a more familiar feel to new users who are used to other SCM systems.

Another popular tool based on git, StGIT, is designed to help a workflow that depends more heavily on exchange of patches. While the primary way to integrate changes from different tracks of development is to make merges in the workflow git and Cogito primarily targets, StGIT supports the workflow to build up piles of patches to be fed upstream, and re-sync with the upstream when some or all of the patches are accepted by rebuilding the patch queue.

While different Porcelains can be used by different people with different work habits, the development history recorded by different Porcelains are eventually made by the common Plumbing commands in the same underlying format, and therefore are compatible with each other. This allows people with different workflow and choice of tools to cooperate on the same project.

5 Weakness and Future Works

There are various missing features and unfinished parts in the current system that require further improvements. Note that the system is still evolving at a rapid pace and some of the issues listed here may have already been addressed when this paper is published.

5.1 Partial history

The system operates on commit ancestry chains to perform many of the interesting things it does, and most of the time it only needs to look at the commits near the tip of the branches. An obvious example is to look at recent development histories. Merging branches need access to the commits on the ancestry chain down to the latest common ancestor commit, and no earlier history is required. One thing that is often desired but not currently supported is to make a "shallow" clone of a repository that records only the recent history, and later deepen it by retrieving older commits.

Synchronizing two git repositories is done by comparing the heads of branches on both repositories, finding common ancestors and copying the commits and their associated tree and blob objects that are missing from one end to the other. This operation relies on an invariant that all history behind commits that are recorded as the heads of branches are already in the repository. Making a shallow clone that has only commits near the tip of the branch violates this invariant, and a later attempt to download older history would become a no-operation. To support "shallow" cloning, this invariant needs to be conditionally lifted during "history deepening" operation.

5.2 Subprojects

Multiple projects overlayed in a single directory are not supported. Different repositories can be stored along with their associated working trees in separate subdirectories, but currently there is no support to tie the versions from the different subprojects together.

There have been discussions on this topic and two alternative approaches were proposed, but there were not enough interest to cause either approach to materialize in the form of concrete code yet.

5.3 Implications of not recording renames

In an early stage of the development, we decided not to record rename information in trees nor commits. This was both practical and philosophical.

Files are not renamed that often, and it was observed that moving file contents around without moving the file itself happened just as often. Not having to record renames specially, but always recording the state of the whole tree in each revision, was easier to implement from a practical point of view.

When examining the project history, the question "*where did this function come from, and how did it get into the current form?*" is far more interesting than "*where did this file come from?*" and when the former question is answered properly (i.e. "*it started in this shape in file* X*, but later assumed that shape and migrated to file* Y"), the latter becomes a narrow special case, and we did not want to only support the special case. Instead, we wanted to solve the former problem in a way general enough to make the latter a non-issue. However, deliberately not recording renames often contradicts people's expectations.

Currently we have a merge strategy that looks at the common ancestor and two branch heads being merged to detect file renames and try to merge the contents accordingly. If the modifications made to the file in question across renames is too big, the rename detection logic would not notice that they are related. It is possible for the merge algorithm to inspect all commits along the ancestry chain to make the rename detection more precise, but this would make merges more expensive.

6 Conclusion

Git started as a necessity to have a minimally usable system, and during its brief development history, it has quickly become capable of hosting one of the most important free software projects, the Linux kernel. It is now used by projects other than the kernel (to name a few: Cairo, Gnumeric, Wine, xmms2, the X.org X server). Its simple model and tool-based approach allow it to be enhanced to support different workflows by scripting around it and still be compatible with other people who use it. The development community is active and is growing (about 1500 postings are made to the mailing list every month).

Reducing fsck time for ext2 file systems

Val Henson
Intel, Inc.
`val_henson@intel.com`

Zach Brown
Oracle, Inc.
`zach.brown@oracle.com`

Theodore Ts'o
IBM, Inc.
`tytso@alum.mit.edu`

Arjan van de Ven
Intel, Inc.
`arjan@linux.intel.com`

Abstract

Ext2 is fast, simple, robust, and fun to hack on. However, it has fallen out of favor for one major reason: if an ext2 file system is not cleanly unmounted, such as in the event of kernel crash or power loss, it must be repaired using `fsck`, which takes minutes or hours to complete, during which time the file system is unavailable. In this paper, we describe some techniques for reducing the average fsck time on ext2 file systems. First, we avoid running fsck in some cases by adding a filesystem-wide dirty bit indicating whether the file system was being actively modified at the time it crashed. The performance of ext2 with this change is close to that of plain ext2, and quite a bit faster than ext3. Second, we propose a technique called linked writes which uses dependent writes and a list of dirty inodes to allow recovery of an active file system by only repairing the dirty inodes and avoiding a full file system check.

1 Introduction

The Second Extended File System, ext2, was implemented in 1993 by Remy Card, Theodore T'so, and Stephen Tweedie, and for many years was the file system of choice for Linux systems. Ext2 is similar in on-disk structure to the Berkeley FFS file system [14], with the notable exception of sub-block size fragments [2]. In recent years, ext2 has been overtaken in popularity by the ext3 [6, 16] and reiser3 [4] file systems, both journaling file systems. While these file systems are not as fast as ext2 in some cases [1], and are certainly not as simple, their recovery after crash is very fast as they do not have to run fsck.

Like the original Berkeley FFS, ext2 file system consistency is maintained on a post hoc basis, by repair after the fact using the file system checker, fsck [12]. Fsck works by traversing the entire file system and building up a consistent picture of the file system metadata, which it then writes to disk. This kind of post hoc data repair has two major drawbacks. One, it tends to be fragile. A new set of test and repair functions had to be written for every common kind of corruption. Often, fsck had to fall back to manual mode—that is, asking the human to make decisions about repairing the file system for it. As ext2 continued to be used and new tests and repairs were added to the fsck code base, this occurred less and less often, and now most users can reasonably expect fsck to complete unattended after a system crash.

The second major drawback to fsck is total running time. Since fsck must traverse the entire file system to build a complete picture of allocation bitmaps, number of links to inodes, and other potentially incorrect metadata, it takes anywhere from minutes to hours to complete. File system repair using fsck takes time proportional to the size of the file system, rather than the size of the ongoing update to the file system, as is the case for journaling file systems like ext3 and reiserfs. The cost of the system unavailability while fsck is running is so great that ext2 is generally only used in niche cases, when high ongoing performance is worth the cost of occasional system unavailability and possible greater chance of data loss.

On the other hand, ext2 is fast, simple, easy to repair, uses little CPU, performs well with multi-threaded reads and writes, and benefits from over a decade of debugging and fine tuning. Our goal is to find a way to keep these attributes while reducing the average time it takes to recover from crashes—that is, reducing the average time spent running fsck. Our target use case is a server with many users, infrequent writes, lots of read-only file system data, and tolerance for possibly greater chance of data loss.

Our first approach to reducing fsck time is to implement a filesystem-wide dirty bit. While writes are in progress, the bit is set. After the file system has been idle for some period of time (one second in our implementation), we force out all outstanding writes to disk and mark the file system as clean. If we crash while the file system is marked clean, fsck knows that it does not have to do a full fsck. Instead, it does some minor housekeeping and marks the file system as valid. Orphan inodes and block preallocation added some interesting twists to this solution, but overall it remains a simple change. While this approach does not improve worst case fsck time, it does improve average fsck time. For comparison purposes, recall that ext3 runs a full fsck on the file system every 30 mounts as usually installed.

Our second approach, which we did not implement, is an attempt to limit the data fsck needs to examine to repair the file system to a set of dirty inodes and their associated metadata. If we add inodes to an on-disk dirty inode list before altering them and correctly order metadata writes to the file system, we will be able to correct allocation bitmaps, directory entries, and inode link counts without rebuilding the entire file system, as fsck does now.

Some consistency issues are difficult to solve without unsightly and possibly slow hacks, such as keeping the number of links consistent for a file with multiple hard links during an `unlink()` operation. However, they occur relatively rarely, so we are considering combining this approach with the filesystem-wide dirty bit. When a particular operation is too ugly to implement using the dirty inode list, we simply mark the file system as dirty for the duration of the operation. It may be profitable to merely narrow the window during which a crash will require a full fsck rather than to close the window fully. Whether this can be done and still preserve the properties of simplicity of implementation and high performance is an open question.

2 Why ext2?

Linux has a lot of file systems, many of which have better solutions for maintaining file system consistency than ext2. Why are we working on improving crash recovery in ext2 when so many other solutions exist? The answer is a combination of useful properties of ext2 and drawbacks of existing file systems.

First, the advantages of ext2 are simplicity, robustness, and high performance. The entire ext2 code base is about 8,000 lines of code; most programmers can understand and begin altering the codebase within days or weeks. For comparison, most other file systems come in anywhere from 20,000 (ext3 + jbd) to 80,000 (XFS) lines of code. Ext2 has been in active use since 1993, and benefits from over a decade of weeding out bugs and repairing obscure and seldom seen failure cases. Ext2 performance is quite good overall, especially considering its simplicity, and definitely superior to ext3 in most cases.

The main focus of file systems development in Linux today is ext3. On-disk, ext3 is almost identical to ext2; both file systems can be mounted as either ext2 or ext3 in most cases. Ext3 is a journalled file system; updates to the file system are first written as compact entries in the on-disk journal region before they are written to their final locations. If a crash occurs during an update, the journal is replayed on the next mount, completing any unfinished updates.

Our primary concern with ext3 is lower performance from writing and sharing the journal. Work is being done to improve performance, especially in the area of multi-threaded writes [9], but it is hard to compete in performance against a file system which has little or no restrictions in terms of sharing resources or write ordering. Our secondary concern is complexity of code. Journaling adds a whole layer of code to open transactions, reserve log space, and bail out when an error occurs. Overall, we feel that ext3 is a good file system for laptops, but not very good for write-intensive loads.

The reiser3 [4] file system is the default file system for the SuSE distribution. It is also a journaling file system, and is especially good for file systems with many small files because it packs the file together, saving space. The performance of reiser3 is good and in some cases better than ext2. However, reiser3 was developed outside the mainstream Linux community and never attracted a community developer base. Because of this and the complexity of the implementation, it is not a good base for file system development. Reiser4 [4] has less developer buy-in, more code, worse performance in many cases [1], and may not be merged into the mainline Linux tree at all [3].

XFS is another journaling file system. It has many desirable properties, and is ideal for applications requiring thousands or millions of files in one directory, but also suffers from complexity and lack of developer community. Performance of more common case operations (such as file create) suffers for the benefit of fast look ups in directories with many entries [1].

Other techniques for maintaining file system complexity are soft updates [13] and copy-on-write [11]. Without a team of full-time programmers and several years to work on the problem, we did not feel we could implement either of these techniques. In any case, we did not feel we could maintain the simplicity or the benefits of more than a decade of testing of ext2 if we used these techniques.

Given these limitations, we decided to look for "90% solutions" to the file system consistency problem, starting with the ext2 code base. This paper describes one technique we implemented, the filesystem-wide dirty bit, and one we are considering implementing, linked writes.

3 The fsck program

Cutting down crash recovery time for an ext2 file system depends on understanding how the file system checker program, fsck works. After Linux has finished booting the kernel, the root file system is mounted read-only and the kernel executes the init program. As part of normal system initialization, fsck is run on the root file system before it is remounted read-write and on other file systems before they are mounted. Repair of the file system is necessary before it can be safely written.

When fsck runs, it checks to see if the ext2 file system was cleanly unmounted by reading the state field in the file system superblock. If the state is set as VALID, the file system is already consistent and does not need recovery; fsck exits without further ado. If the state is INVALID, fsck does a full check of the file system integrity, repairing any inconsistencies it finds. In order to check the correctness of allocation bitmaps, file nlinks, directory entries, etc., fsck reads every inode in the system, every indirect block referenced by an inode, and every directory entry. Using this information, it builds up a new set of inode and block allocation bitmaps, calculates the correct number of links of every inode, and removes directory entries to unreferenced inodes. It does many other things as well, such as sanity check inode fields, but these three activities fundamentally require reading every inode in the file system. Otherwise, there is no way to find out whether, for example, a particular block is referenced by a file but is marked as unallocated on the block allocation bitmap. In summary, there are no back pointers from a data block to the indirect block that points to it, or from a file to the directories that point to it, so the only way to reconstruct reference counts is to start at the top level and build a complete picture of the file system metadata.

Unsurprisingly, it takes fsck quite some time to rebuild the entirety of the file system metadata, approximately O(total file system size + data stored). The average laptop takes several minutes to fsck an ext2 file system; large file servers can sometimes take hours or, on occasion, days! Straightforward tactical performance optimizations such as requesting reads of needed blocks in sequential order and read-ahead requests can only improve the situation so much, given that the whole operation will still take time proportional to the entire file system. What we want is file system recovery time that is O(writes in progress), as is the case for journal replay in journaling file systems.

One way to reduce fsck time is to eliminate the need to do a full fsck at all if a crash occurs when the file system is not being changed. This is the approach we took with the filesystem-wide dirty bit.

Another way to reduce fsck time is to reduce the amount of metadata we have to check in order to repair the file system. We propose a method of ordering updates to the file system in such a way that full consistency can be recovered by scanning a list of dirty inodes.

4 Implementation of filesystem-wide dirty bit

Implementing the fs-wide dirty bit seemed at first glance to be relatively simple. Intuitively, if no writes are going on in the file system, we should be able to sync the file system (make sure all outstanding writes are on disk), reset the machine, and cleanly mount the unchanged file system. Our intuition is wrong in two major points: orphan inodes, and block preallocation. Orphan inodes are files which have been unlinked from the file system, but are still held open by a process. On crash and recovery, the inode and its blocks need to be freed. Block preallocation speeds up block allocation by preallocating a few more blocks than were actually requested. Unfortunately, as implemented, preallocation alters on-disk data, which needs to be corrected if the file is not cleanly closed. First we'll describe the overall implementation, then our handling of orphan inodes and preallocated blocks.

4.1 Overview of dirty bit implementation

Our first working patch implementing the fs-wide dirty bit included the following high-level changes:

- Per-mount kernel thread to mark file system clean
- New `ext2_mark_*_dirty()` functions
- Port of ext3 orphan inode list
- Port of ext3 reservation code

The ports of the ext3 orphan inode list and reservation code were not frivolous; without them, the file system would be an inconsistent state even when no writes were occurring without them.

4.2 Per-mount kernel thread

The basic outline of how the file system is marked dirty or clean by the per-mount kernel thread is as follows:

- Mark the file system dirty whenever metadata is altered.
- Periodically check the state of the file system.
- If the file system is clean, sync the file system.
- If no new writes occurred during the sync, mark the file system clean.

The file system is marked clean or dirty by updating a field in the superblock and submitting the I/O as a barrier write so that no writes can pass it and hit the disk before the dirty bit is updated. The update of the dirty bit is done asynchronously, so as to not stall during the first write to a clean file system (since it is a barrier write, waiting on it will not change the order of writes to disk anyway).

In order to implement asynchronous update of the dirty bit in the superblock, we needed to create an in-memory copy of the superblock. Updates to the superblock are written to the in-memory copy; when the superblock is ready to be written to disk, the superblock is locked, the in-memory superblock is copied to the buffer for the I/O operation, and the I/O is submitted. The code implementing the superblock copy is limited to the files `ext2/super.c` and one line in `ext2/xattr.c`.

One item on our to-do list is integration with the laptop mode code, which tries to minimize the number of disk spin-up and spin-down events by concentrating disk write activity into batches. Marking the file system clean should probably be triggered by the timeout for flushing dirty data in laptop mode.

4.3 Marking the file system dirty

Before any metadata changes are scheduled to be written to disk, the file system must first be marked dirty. Ext2 already uses the functions `mark_inode_dirty()`, `mark_buffer_dirty()`, and `mark_buffer_dirty_inode()` to mark changed metadata for write-out by the VFS and I/O subsystems. We created ext2-specific versions of these functions which first mark the file system dirty and then call the original function.

4.4 Orphan inodes

The semantics of UNIX file systems allow an application to create a file, open it, unlink the file (removing any reference to it from the file system), and keep the file open indefinitely. While the file is open, the file system can not delete the file. In effect, this creates a temporary file which is guaranteed to be deleted, even if the system crashes. If the system does crash while the file is still open, the file system contains an orphan inode—an inode which marked as in use, but is not referenced by any directory entry. This behavior is very convenient for application developers and a real pain in the neck for file system developers, who wish they would all use files in tmpfs instead.

In order to clean up orphan inodes after a crash, we ported the ext3 orphan inode list to ext2. The orphan inode list is an on-disk singly linked list of inodes, beginning in the orphan inode field of the superblock. The `i_dtime` field of the inode, normally used to store the time an inode was deleted, is (ab)used as the inode number of the next item in the orphan inode list. When fsck is run on the file system, it traverses the linked list of orphan inodes and frees them. Fortunately for us, the code in fsck that does this runs regardless of whether the file system is mounted as ext2 or ext3.

Our initial implementation followed the ext3 practice of writing out orphan inodes immediately in order to keep the orphan inode list as up-to-date as possible on disk. This is expensive, and an up-to-date orphan inode list is superfluous except when the file system is marked clean. We modified the orphan inode code to only maintain the orphan inode list in memory, and write it out to disk on file system sync. We will need to add a patch to keep fsck from complaining about a corrupted orphan inode list.

4.5 Preallocated blocks

The existing code in ext2 for preallocating blocks unfortunately alters on-disk metadata, such as the block group free and allocated block counts. One solution was to simply turn off preallocation. Fortunately, Mingming Cao implemented new block preallocation code for ext3 which reserves blocks without touching on-disk data, and is superior to the ext2 preallocation code in several other ways. We chose to port Mingming Cao's reservation code to ext2, which in theory should improve block allocation anyway. ext2 and ext3 were similar enough that we could complete the port quickly, although porting some parts of the new `get_blocks()` functionality was tricky.

4.6 Development under User-mode Linux

We want to note that the implementation of the filesystem-wide dirty bit was tested almost entirely on User-mode Linux [10], a port of Linux that runs as a process in Linux. UML is well suited to file system development, especially when the developer is limited to a single laptop for both development host and target platform (as is often the case on an airplane). With UML, we could quickly compile, boot, crash, and reboot our UML instance, all without worrying about corrupting any important file systems. When we did corrupt the UML file system, all that was necessary was to copy a clean file system image back over the file containing the UML file system image. The loopback device made it easy to mount, fsck, or otherwise examine the UML file system using tools on the host machine. Only one bug required running on a non-UML system to discover, which was lack of support for suspend in the dirty bit kernel thread.

However, getting UML up and running and working for file system development was somewhat non-intuitive and occasionally baffling. Details about running UML on recent 2.6 kernels, including links to a sample root file system and working `.config` file, can be found here:

`http://www.nmt.edu/~val/uml_tips.html`

5 Performance

We benchmarked the filesystem-wide dirty bit implementation to find out if it significantly impacted performance. On the face of it, we expected a small penalty on the first write, due to issuing an asynchronous write barrier the first time the file system is written.

The benchmarks we ran were *kuntar*, *postmark*, and *tiobench* [7]. Kuntar simply measures the time to extract a cached uncompressed kernel tarball and sync the file system. Postmark creates and deletes many small files in a directory and is a metadata intensive workload. We ran it with `numbers = 10000` and `transactions = 10000`. We also added a `sync()` system call to postmark before the final timing measurement was made, in order to measure the true performance of writing data all the way to the disk. Tiobench is a benchmark designed to measure multi-threaded I/O to a single file; we ran it mainly as a sanity check since we didn't expect anything to change in this workload. We ran tiobench with 16 threads and a 256MB file size.

The file systems we benchmarked were ext2, ext2 with the reservations-only patch, ext2 with reservations only with reservations turned off, ext2 with the fs-wide bit patch and reservations, ext3 with defaults, and ext3 with `data=writeback` mode. All file systems used 4KB blocks and were mounted with the `noatime` option. The kernel was 2.6.16-mm1. The machine had two 1533 MHZ AMD Athlon processors and 1GB of memory. We recored elapsed time, sectors read, sectors written, and kernel ticks. The results are in Table 1.

The results are somewhat baffling, but overall positive for the dirty bit implementation. The times for the fs-wide dirty bit are within 10% of those of plain ext2 for all benchmarks except postmark. For postmark, writes increased greatly for the fs-wide dirty bit; we are not sure why yet. The results for the reservations-only versions of ext2 are even more puzzling; we suspect that our port of reservations is buggy or suboptimal. We will continue researching the performance issues.

We would like to briefly discuss the `noatime` option. All file systems were mounted with the `noatime` option, which turns off updates to the "last accessed time" field in the inode. We turned this option off not only because it would prevent the fs-wide dirty bit from being effective when a file system is under read activity, but also because it is a common technique for improving performance. `noatime` is widely regarded as the correct behavior for most file systems, and in some cases is shipped as the default behavior by distributions. While correct access time is sometimes useful or even critical, such as in tracing which files an intruder read, in most cases it is unnecessary and only adds unnecessary I/O to the system.

6 Linked writes

Our second idea for reducing fsck time is to order writes to the file system such that the file

		ext2	ext2r	ext2rnor	ext2fw	ext3	ext3wb
kuntar	secs	20.32	21.03	19.06	18.87	20.99	32.02
	read	5152	5176	5176	5176	168	168
	write	523272	523272	523288	523304	523256	544160
	ticks	237	269	357	277	413	402
krmtar	secs	9.79	10.92	9.99	10.90	55.64	9.74
	read	20874	20842	20874	20874	20866	20874
	write	5208	5176	5208	5960	36296	10560
	ticks	61	61	62	61	7943	130
postmark	secs	33.98	49.34	42.93	50.46	43.48	41.82
	read	2568	2568	2568	2568	56	48
	write	168312	168392	168392	240720	260704	173936
	ticks	641	650	838	674	1364	1481
tiobench	secs	37.48	35.22	33.68	33.57	35.16	36.69
	read	32	32	32	32	24	112
	write	64	64	64	72	136	136
	ticks	441	450	456	463	452	463

kuntar: expanding a cached uncompressed kernel tarball and syncing
krmtar: rm -rf on cold untarred kernel tree, sync
postmark: postmark + sync() patch, numbers = 10000, transactions = 10000
tiobench: tiobench: 16 threads, 256m size
ext2: ext2
ext2r: ext2, reservations
ext2rnor: ext2, reservations, -o noreservation option
ext2fw: ext2, reservations, fswide
ext3: ext3, 256m journal
ext3wb: ext3, 256m journal, data=writeback

Table 1: Benchmark results

system can repaired to a consistent state after processing a short list of dirty inodes. Before an operation begins, the relevant inodes are added to an on-disk list of dirty inodes. During the operation, we only overwrite references to data (such as indirect blocks or directory entries) after we have finished all updates that require that information (such as updating allocation bitmaps or link counts). If we crash half-way through an operation, we examine each inode on the dirty inode list and repair any consistencies in the metadata it points to. For example, if we were to crash half-way through allocating a block, we would check if each block were marked as allocated in the block allocation bitmap. If it was not, we would free that block from the file (and all blocks that it points to). We call this scheme *linked writes*—a write erasing a pointer is linked or dependent on the write of another block completing first.

Some cases are ambiguous as to what operation was in progress, such as truncating and extending a file. In these cases, we will take the safest action. For example, in an ambiguous truncate/extend, we would assume a truncate operation was in progress, because if we were wrong, the new block would contain uninitialized data, resulting in a security hole. It might be possible to indicate which operation was in progress using other metadata, such as inode size, but if that is not possible or would harm performance, we have this option as a fail safe. The difference between restoring one or the other of two ambiguous operations is the difference between restoring the file as of a short time before the crash versus restoring it as of after the completion of the operation in progress at the time of crash. Either option is allowed; only calling `sync()` defines what state the file is in on-disk at any particular moment.

Some operations may not be recoverable only by ordering writes. Consider removing one hard link to a file with multiple hard links from different directories. The only inodes on the dirty inode list are the inode for the directory we are removing the link from, and the file inode—not the inodes for the other directories with hard links to this file. Say we decrement the link count for the inode, and then crash. In the one link case, when we recover, we will find an inode with link count equal to 0, and a directory with an entry pointing to this inode. Recovery is simple; free the inode and delete the directory entry. But if we have multiple hard links to the file, and the inode has a link count of one or more, we have no way of telling whether the link count was already decremented before we crashed or not. A solution to this is to overwrite the directory entry with an invalid directory entry with a magic record that contains the inode's correct link count which is only replayed if the inode has not already been updated. This regrettably adds yet another linked write to the process of deleting an entry. On the other hand, adding or removing links to files with link counts greater than one is painful but blessedly uncommon. Typically only directories have a link count greater than one, and in modern Linux, directory hard links are not allowed, so a directory's link count can be recalculated simply by scanning the directory itself.

Another problem is circular dependencies between blocks that need to be written out. Say we need to write some part of block A to disk before we write some part of block B. We update the buffers in memory and mark them to be written out in order A, B. But then something else happens, and now we need to write some part of block B to disk before some part of block A. We update the buffers in memory—but now we can't write either block A or block B. Linked writes doesn't run into this problem because (a) every block contains only one kind of metadata, (b) the order in which different kinds of metadata must be written is the same for every option. This is equivalent to the lock ordering solution to the deadlock problem; if you define the order for acquiring locks and adhere to it, you can't get into a deadlock.

Ordinarily, writing metadata in the same order according to type for all operations would not be possible. Consider the case of creating a file versus deleting it. In the create case, we must write the directory entry pointing to the inode before updating the bitmap in order to avoid leaking an inode. In the delete case, we must write the bitmap before we delete the entry to avoid leaking an inode. What gets us out of this circular dependency is the dirty inode list. If we instead put the inode to be deleted on the dirty inode list, then we can delete the directory entry before the bitmap, since if we crash, the inode's presence on the dirty inode list will allow us to update the bitmap correctly. This allows us to define the dependency order "write bitmaps before directory entries." The order of metadata operations for each operation must be carefully defined and adhered to.

When writing a buffer to disk, we need to be sure it does not change in flight. We have two options for accomplishing this: either lock the buffer and stall any operations that need to write to it while it is in flight, or clone the buffer and send the copy to disk. The first option is what soft updates uses [13]; surprisingly performance is quite good so it may be an option. The second option requires more memory but would seem to have better performance.

Another issue is reuse of freed blocks or inodes before the referring inode is removed from the dirty inode list. If we free a block, then reuse it before the inode referring to it is removed from the dirty list, it could be erroneously marked as free again at recovery time. To track this, we need a temporary copy of each affected bitmap showing which items should not be allocated, in addition to the items marked allocated in the main copy of the bitmap. Overall, we occasionally need three copies of each active bitmap in memory. The required memory usage is comparable to that of journaling, copy-on-write, or soft updates.

6.1 Implementing write dependencies

Simply issuing write barriers when we write the first half of a linked write would be terribly inefficient, as the only method of implementing this operation that is universally supported by disks is: (1) issue a cache flush command; (2) issue the write barrier I/O; (3) wait for the I/O to complete; (4) issue a second cache flush command. (Even this implementation may be an illusion; reports of IDE disks which do not correctly implement the cache flush command abound.) This creates a huge bubble in the I/O pipeline. Instead, we want to block only the dependent write. This can be implemented using asynchronous writes which kick off the linked write in the I/O completion handler.

6.2 Comparison of linked writes

Linked writes bears a strong resemblance to soft updates [13]. Indeed, linked writes can be thought of as soft updates from the opposite direction. Soft updates takes the approach of erring on the side of marking things allocated when they are actually free, and then recovering leaked inodes and blocks after mount by running a background fsck on the file system. Linked writes errs on the side of marking things unallocated when they are still referenced by the file system, and repairing inconsistencies by reviewing a list of dirty inodes. Soft updates handles circular buffer dependencies (where block A must be written out before block B and vice versa) by rolling back the dependent data before writing the block out to disk. Linked writes handle circular dependencies by making them impossible.

Linked writes can also be viewed as a form of journaling in which the journal entries are scattered across the disk in the form of inodes and directory entries, and linked together by the dirty inode list. The advantages of linked writes over journaling is that changes are written once, no journal space has to be allocated, writes aren't throttled by journal size, and there are no seeks to a separate journal region.

6.3 Reinventing the wheel?

Why bother implementing a whole new method of file system consistency when we have so many available to us already? Simply put, frustration with code complexity and performance. The authors have had direct experience with the implementation of ZFS [8], ext3 [6], and ocfs2 [5] and were disappointed with the complexity of the implementation. Merely counting lines of code for reiser3 [4], reiser4 [4], or XFS [15] incites dismay. We have not yet encountered someone other than the authors of the original soft updates [13] implementation who claims to understand it well enough to re-implement from scratch. Yet ext2, one of the smallest, simplest file systems out there, continues to be the target for performance on general purpose workloads.

In a sense, ext2 is cheating, because it does not attempt to keep the on-disk data structures intact. In another sense, ext2 shows us what our rock-bottom performance expectations for new file systems should be, as relatively little effort has been put into optimizing ext2.

With linked writes, we hope for a file system a little more complex, a lot more consistent, and nearly the same performance as ext2.

6.4 Feasibility of linked writes implementation

We estimate that implementing linked writes would take on the order of half the effort necessary to implement ext3. Adjusting for programmer capability and experience (translation: I'm no Kirk McKusick or Greg Ganger), we estimate that implementing linked writes would take one fifth the staff-years required by soft updates.

We acknowledge that the design of linked writes is half-finished at best and may end up having fatal flaws, nor do we expect our design to survive implementation without major changes—"There's many a slip 'twixt cup and lip."

7 Failed ideas

Linked writes grew out of our original idea to implement per-block group dirty bits. We wanted to restrict how much of the file system had to be reviewed by fsck after a crash, and dividing it up by block groups seemed to make sense. In retrospect, we realized that the only checks we could do in this case would start with the inodes in this block group and check file system consistency based on the information they point to. On the other hand, given a block allocation bitmap, we can't check whether a particular block is correctly marked unless we rebuild the entire file system by reading all of the inodes. In the end, we realized that per-bg dirty bits would basically be a very coarse hash of which inodes need to be checked. It may make sense to implement some kind of bitmap showing which inodes need to be checked rather than a linked list, otherwise this idea is dead in the water.

Another idea for handling orphan inodes was to implement a set of "in-memory-only" bitmaps that record inodes and blocks which are allocated only for the lifetime of this mount—in other words, orphan inodes and their data. However, these bitmaps would in the worst case require two blocks per cylinder group of unreclaimable memory. A workaround would be to allocate space on disk to write them out under memory pressure, but we abandoned this idea quickly.

8 Availability

The most recent patches are available from:

`http://www.nmt.edu/~val/patches.html`

9 Future work

The filesystem-wide dirty bit seems worthwhile to polish for inclusion in the mainline kernel, perhaps as a mount option. We will continue to do work to improve performance and test correctness.

Implementing linked writes will take a significant amount of programmer sweat and may not be considered, shall we say, business-critical to our respective employers. We welcome discussion, criticism, and code from interested third parties.

10 Acknowledgments

Many thanks to all those who reviewed and commented on the initial patches. Our work was greatly reduced by being able to port the orphan inode list from ext3, written by Stephen Tweedie, as well as the ext3 reservation patches by Mingming Cao.

11 Conclusion

The filesystem-wide dirty bit feature allows ext2 file systems to skip a full fsck when the file system is not being actively modified during a crash. The performance of our initial, untuned implementation is reasonable, and will be improved. Our proposal for linked writes outlines a strategy for maintaining file system consistency with less overhead than journaling and simpler implementation than copy-on-write or soft updates.

We take this opportunity to remind file system developers that ext2 is an attractive target for innovation. We hope that developers rediscover the possibilities inherent in this simple, fast, extendable file system.

References

[1] Benchmarking file systems part II LG #122. `http://linuxgazette.net/122/piszcz.html`.

[2] Design and implementation of the second extended filesystem. `http://e2fsprogs.sourceforge.net/ext2intro.html`.

[3] Linux: Reiser4 and the mainline kernel. `http://kerneltrap.org/node/5679`.

[4] Namesys. `http://www.namesys.com/`.

[5] OCFS2. `http://oss.oracle.com/projects/ocfs2/`.

[6] Red hat's new journaling file system: ext3. `http://www.redhat.com/support/wpapers/redhat/ext3/`.

[7] Threaded I/O tester. `http://sourceforge.net/projects/tiobench`.

[8] ZFS at OpenSolaris.org. `http://www.opensolaris.org/os/community/zfs/`.

[9] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Ottawa Linux Symposium 2005*, July 2005.

[10] Jeff Dike. User-mode linux. In *Ottawa Linux Symposium 2001*, July 2001.

[11] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an nfs file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[12] T. J. Kowalski and Marshall K. McKusick. Fsck - the UNIX file system check program. Technical report, Bell Laboratories, March 1978.

[13] Marshall K. McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17. USENIX, 1999.

[14] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[15] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Michael Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Technical Conference*, 1996.

[16] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, 1998.

Native POSIX Threads Library (NPTL) Support for uClibc.

Steven J. Hill
Reality Diluted, Inc.
sjhill@realitydiluted.com

Abstract

Linux continues to gain market share in embedded systems. As embedded processing power increases and more demanding applications in need of multi-threading capabilities are developed, Native POSIX Threads Library (NPTL) support becomes crucial. The GNU C library [1] has had NPTL support for a number of years on multiple processor architectures. However, the GNU C library is more suited for workstation and server platforms and not embedded systems due to its size. uClibc [2] is a POSIX-compliant C library designed for size and speed, but currently lacking NPTL support. This paper will present the design and implementation of NPTL support in uClibc. In addition to the design overview, benchmarks, limitations and comparisons between glibc and uClibc will be discussed. NPTL for uClibc is currently only supported for the MIPS processor architecture.

1 The Contenders

Every usable Linux system has applications built atop a C library run-time environment. The C library is at the core of user space and provides all the necessary functions and system calls for applications to execute. Linux is fortunate in that there are a number of C libraries available for varying platforms and environments. Whether an embedded system, high-performance computing, or a home PC, there is a C library to fit each need.

The GNU C library, known also as glibc [1], and uClibc [2] are the most common Linux C libraries in use today. There are other C libraries like Newlib [3], diet libc [4], and klibc [5] used in embedded systems and small root file systems. We list them only for completeness, yet they are not considered in this paper. Our focus will be solely on uClibc and glibc.

2 Comparing C Libraries

To understand the need for NPTL in uClibc, we first examine the goals of both the uClibc and glibc projects. We will quickly examine the strengths and weaknesses of both C implementations. It will then become evident why NPTL is needed in uClibc.

2.1 GNU C Library Project Goals

To quote from the main GNU C Library web page [1], "The GNU C library is primarily designed to be a portable and high performance C library. It follows all relevant standards (ISO C 99, POSIX.1c, POSIX.1j, POSIX.1d, Unix98, Single Unix Specification). It is also internationalized and has one of the most complete internationalization interfaces known." In short, glibc, aims to be the most complete C library implementation available. It succeeds, but at the cost of size and complexity.

2.2 uClibc Project Goals

Let us see what uClibc has to offer. Again, quoting from the main page for uClibc [2], "uClibc (a.k.a. μClibc, pronounced *yew-see-lib-see*) is a C library for developing embedded Linux systems. It is much smaller than the GNU C Library, but nearly all applications supported by glibc also work perfectly with uClibc. Porting applications from glibc to uClibc typically involves just recompiling the source code. uClibc even supports shared libraries and threading. It currently runs on standard Linux and MMU-less (also known as μClinux) systems..." Sounds great for embedded systems development. Obviously, uClibc is going to be missing some features since its goal is to be small in size. However, uClibc has its own strengths as well.

2.3 Comparing Features

Table 1 shows the important differentiating features between glibc and uClibc.

It should be obvious from the table above that glibc certainly has better POSIX compliance, backwards binary compatibility, and networking services support. uClibc shines in that it is much smaller (how much smaller will be covered later), more configurable, supports more processor architectures and is easier to build and maintain.

The last two features in the table warrant additional explanation. glibc recently removed `linuxthreads` support from its main development tree. It was moved into a separate ports tree. It is maintained on a volunteer basis only. uClibc will maintain both the `linuxthreads` and `nptl` thread models actively. Secondly, glibc only supports a couple of primary processor architectures. The rest of the architectures were also recently moved into the ports tree. uClibc continues to actively support many more architectures by default. For embedded systems, uClibc is clearly the winner.

3 Why NPTL for Embedded Systems?

uClibc supports multiple thread library models. What are the shortcomings of `linuxthreads`? Why is `nptl` better, or worse? The answer lies in the requirements, software and hardware, of the embedded platform being developed. We need to first compare `linuxthreads` and `nptl` to choose the thread library that best meets the needs of our platform. Table 2 lists the key features the two thread libraries have to offer.

Using the table above, the `nptl` model is useful in systems that do not have severe memory constraints, but need threads to respond quickly and efficiently. The `linuxthreads` model is useful mostly for resource constrained systems still needing basic thread support. As mentioned at the beginning of this paper, embedded systems with faster processors and greater memory resources are being required to do more, with less. Using NPTL in conjunction with the already small C library provided by uClibc, creates a perfectly balanced embedded Linux system that has size, speed and an high-performance multi-threading.

Feature	glibc	uClibc
LGPL	Y	Y
Complete POSIX compliance	Y	N
Binary compatibility across releases	Y	N
NSS Support	Y	N
NIS Support	Y	N
Locale support	Y	Y
Small disk storage footprint	N	Y
Small runtime memory footprint	N	Y
Supports MMU-less systems	N	Y
Highly configurable	N	Y
Simple build system	N	Y
Built-in configuration system	N	Y
Easily maintained	N	Y
NPTL Support	Y	Y
Linuxthreads Support	N (See below)	Y
Support many processor architectures	N (See below)	Y

Table 1: Library Feature Comparison

Feature	Description	LinuxThreads	NPTL
Storage Size	The actual amount of storage space consumed in the file system by the libraries.	Smallest	Largest
Memory Usage	The actual amount of RAM consumed at run-time by the thread library code and data. Includes both kernel and user space memory usage.	Smallest	Largest
Number of Threads	The maximum number of threads available in a process.	Hard Coded Value	Dynamic
Thread Efficiency	Rate at which threads are created, destroyed, managed, and run.	Slowest	Fastest
Per-Thread Signals	Signals are handled on a per-thread basis and not the process.	No	Yes
Inter-Thread Synchronization	Threads can share synchronization primitives like mutexes and semaphores.	No	Yes
POSIX.1 Compliance	Thread library is compliant	No	

Table 2: Thread Library Features

4 uClibc NPTL Implementation

The following sections outline the major technical components of the NPTL implementation for uClibc. For the most part, they should apply equally to glibc's implementation except where noted. References to additional papers and information are provided should the reader wish to delve deeper into the inner workings of various components.

4.1 TLS—The Foundation of NPTL

The first major component needed for NPTL on Linux systems is Thread Local Storage (TLS). Threads in a process share the same virtual address space. Usually, any static or global data declared in the process is visible to all threads within that process. TLS allows threads to have their own local static and global data. An excellent paper, written by Ulrich Drepper, covers the technical details for implementing TLS for the ELF binary format [6]. Supporting TLS required extensive changes to binutils [7], GCC [8] and glibc [1]. We cover the changes made in the C library necessary to support TLS data.

4.1.1 The Dynamic Loader

The dynamic loader, also affectionately known as `ld.so`, is responsible for the run-time linking of dynamically linked applications. The loader is the first piece of code to execute before the main function of the application is called. It is responsible for loading and mapping in all required shared objects for the application.

For non-TLS applications, the process of loading shared objects and running the application is trivial and straight-forward. TLS data types complicate dynamic linking substantially. The loader must detect any TLS sections, allocate initial memory blocks for any needed TLS data, perform initial TLS relocations, and later perform additional TLS symbol look-ups and relocations during the execution of the process. It must also deal with the loading of shared objects containing TLS data during program execution and properly allocate and relocate its data. The TLS paper [6] provides ample overview of how these mechanisms work. The document does not, however, currently cover the specifics of MIPS-specific TLS storage and implementation. MIPS TLS information is available from the Linux/MIPS website [9].

Adding TLS relocation support into uClibc's dynamic loader required close to 2200 lines of code to change. The only functionality not available in the uClibc loader is the handling of TLS variables in the dynamic loader itself. It should also be noted that statically linked binaries using TLS/NPTL are not currently supported by uClibc. Static binaries will not be supported until all processor architectures capable of supporting NPTL have working shared library support. In reality, shared library support of NPTL is a prerequisite for debugging static NPTL support. Thank you to Daniel Jacobowitz for pointing this out.

4.1.2 TLS Variables in uClibc

There are four TLS variables currently used in uClibc. The noticeable difference that can be observed in the source code is that they have an additional type modifier of `__thread`. Table 3 lists the TLS in detail. There are 15 additional TLS variables for locale support that were not ported from glibc. Multi-threaded locale support with NPTL is currently not supported with uClibc.

Variable	Description
errno	The number of the last error set by system calls and some functions in the library. Previously it was thread-safe, but still shared by threads in the same process. For NPTL, it is a TLS variable and thus each thread has its own instantiation.
h_errno	The error return value for network database operations. This variable was also previously thread safe. It is only available internally to uClibc.
_res	The resolver context state variable for host name look-ups.
RPC_VARS	Pointer to internal Remote Procedure Call (RPC) structure for multi-threaded applications.

Table 3: TLS Variables in uClibc

4.2 Futexes

Futexes [10] [11] are fast user space mutexes. They are an important part of the locking necessary for a responsive pthreads library implementation. They are supported by Linux 2.6 kernels and no code porting was necessary for them to be usable in uClibc other than adding prototypes in a header file. glibc also uses them extensively for the file I/O functions. Futexes were ported for use in uClibc's I/O functions as well, although not strictly required by NPTL. Futex I/O support is a configurable for uClibc and can be selected with the `UCLIBC_HAS_STDIO_FUTEXES` option.

4.3 Asynchronous Thread Cancellation

A POSIX compliant thread library contains the function `pthread_cancel`, which cancels the execution of a thread. Please see the official definition of this function at The Open Group website [12]. Cancellation of a thread must be done carefully and only at certain points in the library. There are close to 40 functions where thread cancellation must be checked for and possibly handled. Some of these are heavily used functions like `read`, `write`, `open`, `close`, `lseek` and others. Extensive changes were made to uClibc's C library core in order to support thread cancellation. A list of these are available on the NPTL uClibc development site [13].

4.4 Threads Library

The code in the nptl directory of glibc was initially copied verbatim from a snapshot dated 20050823. An entirely new set of build files were created in order to build NPTL within uClibc. Almost all of the original files remain with the exception of Asynchronous I/O (AIO) related code. All backwards binary compatibility code and functions have been removed. Any code that was surrounded with `#ifdef SHLIB_COMPAT` or associated with those code blocks was also removed. The uClibc NPTL implementation should be as small as possible and not constrained by old thread compatibility code. This also means that any files in the root `nptl` directory with the prefix of `old_pthread_` were also removed. Finally, there were minor header files changes and some functions renamed.

4.5 POSIX Timers

In addition to the core threads library, there were also code changes for POSIX Timers. These changes were integrated into the `librt` library in uClibc. These changes were not in the original statement of work from Broadcom, but were implemented for completeness. All the timer tests for POSIX timers associated with NPTL do pass, but were not required.

For those interested in further details of the NPTL design, please refer to Ulrich Drepper's design document [14].

5 uClibc NPTL Testing

There were a total of four test suites that the uClibc NPTL implementation was tested against. Hopefully, with all of these tests, uClibc NPTL functionality should be at or near the production quality of glibc's implementation.

5.1 uClibc Testsuite

uClibc has its own test suite distributed with the library source. While there are many tests verifying the inner workings of the library and the various subsystems, pthreads tests are minimal. There are only 7 of them, with another 5 for the dynamic loader. With the addition of TLS data and NPTL, these were simply not adequate for testing the new functionality. They were, however, useful as regression testing. The test suite can be retrieved with uClibc from the main site [2].

5.2 glibc Testsuite

The author would first like to convey immense appreciation to the glibc developers for creating such a comprehensive and usable test suite for TLS and NPTL. Had it not been for the tests distributed with their code, I would have released poor code that would have resulted in customer support nightmares. The tests were very well designed and extremely helpful in finding holes in the uClibc NPTL implementation. 182 selected NPTL tests and 15 TLS tests were taken from glibc and passed successfully with uClibc. There were a number of tests not applicable to uClibc's NPTL implementation that were omitted. For further details concerning the tests, please visit the uClibc NPTL project website [13].

5.3 Linux Test Project

The Linux Test Project (LTP) suite [15] is used to "validate the reliability, robustness, and stability of Linux." It has 2900+ tests that will not only test pthreads, but act as a large set of regression tests to make sure uClibc is still functioning properly as a whole.

5.4 Open POSIX Test Suite

To quote from the website [16], "The POSIX Test Suite is an open source test suite with the goal of performing conformance, functional, and stress testing of the IEEE 1003.1-2001 System Interfaces specification in a manner that is agnostic to any given implementation." This suite of tests is the most important indicator of how correct the uClibc NPTL implementation actually is. It tests pthreads, timers, asynchronous I/O, message queues and other POSIX related APIs.

5.5 Hardware Test Platform

All development and testing was done with an AMD Alchemy DBAu1500 board graciously

Source	Version
binutils	2.16.1
gcc	4.1.0
glibc	20050823
uClibc-nptl	20060318
Linux Kernel Headers	2.6.15
LTP	20050804
Open POSIX Test Suite (Distributed w/LTP)	20050804
buildroot	20060328
crosstool	0.38
Linux/MIPS Kernel	2.6.15

Table 4: Source and Tool Versions

donated by AMD. Broadcom also provided their own hardware, but it was not ready for use until later in the software development cycle.

The DBAu1500 development board is designed around a 400MHz 32-bit MIPS Au1500 processor core. The board has 64MB of 100MHz SDRAM and 32MB of AMD MirrorBit Flash memory. The Au1500 utilizes the MIPS32 Instruction Set and has a 16KB Instruction and 16KB Data Cache. (2) 10/100Mbit Ethernet Ports, USB host controller, PCI 2.2 compliant host controller, and other peripherals.

5.6 Software Versions

Table 4 lists the versions of all the sources used in the development and testing of uClibc NPTL. buildroot [17] was the build system used to create both the uClibc and glibc root filesystems necessary for running the test suites. crosstool [18] was used for building the glibc NPTL toolchain.

The actual Linux kernel version used on the AMD development board for testing is the released Linux/MIPS 2.6.15 kernel. The 2.6.16 release is not currently stable enough for testing and development. A number of system calls appear to be broken along with serial text console responsiveness. The root filesystem was mounted over NFS.

glibc	uClibc
53m 8.983s	21m 33.129s

Table 5: Toolchain Build Times

6 uClibc NPTL Test Results

6.1 Toolchain Build Time

Embedded system targets do not usually have the processor and/or memory resources available to host a complete development environment (compiler, assembler, linker, etc). Usually development is done on an x86 host and the binaries are cross compiled for the target using a cross development toolchain. crosstool [18] was used to build the x86 hosted MIPS NPTL toolchain using glibc, and buildroot [17] was used to build the MIPS NPTL toolchain using uClibc.

Building a cross development toolchain is a time consuming process. Not only are they difficult to get working properly, they also take a long time to build. glibc itself usually must be built twice in order to get internal paths and library dependencies to be correct. uClibc on the other hand, need only be built once due to its simpler design and reduced complexity of the build system. The toolchains were built and hosted on a dual 248 Opteron system with 1GB of RAM, Ultra 160 SCSI and SATA hard drives. Times include the actual extraction of source from the tarballs for toolchain components. See Table 5. The toolchains above compile both C and C++ code for the MIPS target processor. Not only are uClibc's libraries smaller (as you will see shortly), but creating a development environment is much simpler and less time consuming.

6.2 Library Code Size

Throughout our discussion, we have stressed the small size of uClibc as compared to glibc. Tables 6 and 7 show these comparisons of libraries and shared objects as compared to one another.

The size difference between the C libraries is dramatic. uClibc is better than 2 times smaller than glibc. uClibc's `libdl.a` is larger because some TLS functions only used for shared objects are being included in the static library. This is due to a problem in the uClibc build system that will be addressed. The `libnsl.a` and `libresolv.a` libraries are dramatically smaller for uClibc only because they are stub libraries. The functions usually present in the corresponding glibc libraries are contained inside uClibc. Finally, `libm.a` for uClibc is much smaller do to reduced math functionality in uClibc as compared to glibc. Most functions for handling `double` data types are not present for uClibc.

The shared objects of most interest are `libc.so`, `ld.so`, and `libpthread.so`. uClibc's main C library is over **2 times smaller** than glibc. The dynamic loader for uClibc is **4 times smaller**. glibc's dynamic loader is complex and larger, but it has to be in order to handle binary backwards compatibility. Additionally, uClibc's dynamic loader is cannot be executed as an application like glibc's. Although the NPTL pthread library code was ported almost verbatim from glibc, uClibc's library is **30% smaller**. Why? The first reason is that any backward binary compatibility code was removed. Secondly, the comments in the nptl directory of glibc say that the NPTL code should be compiled with the `-O2` compiler option. For uClibc, the `-Os` option was used to reduced the code size and yet NPTL still functioned perfectly. This optimization worked for MIPS, but other architectures may not be able to use this optimization.

6.3 glibc NPTL Library Test Results

The developers of NPTL for glibc created a large and comprehensive test suite for testing its functionality. 182 tests were taken from glibc and tested with uClibc's TLS and NPTL implementation. All of these tests passed with uClibc NPTL. For a detailed overview of the selected tests, please visit the uClibc NPTL project website.

6.4 Linux Test Project (LTP) Results

Out of over 2900+ tests executed, there were only 31 failed tests by uClibc. glibc also failed 31 tests. A number of tests that passed with uClibc, failed with glibc. The converse was also true. These differences will be examined at a later date. However, passing all the tests in the LTP is a goal for uClibc. Detailed test logs can be obtained from the uClibc NPTL project website.

6.5 Open POSIX Testsuite Results

Table 8 shows the results of the test runs for both libraries.

The first discrepancy observed is the total number of tests. glibc has a larger number of tests available because of Asynchronous I/O support and the `sigqueue` function, which is not currently available in uClibc. Had these features been present in uClibc, the totals would have

Static Library	**glibc [bytes]**	**uClibc [bytes]**
`libc.a`	3 426 208	1 713 134
`libcrypt.a`	29 154	15 630
`libdl.a`	10 670	36 020
`libm.a`	956 272	248 598
`libnsl.a`	161 558	1 100
`libpthread.a`	281 502	250 852
`libpthread_nonshared.a`	1 404	1 288
`libresolv.a`	111 340	1 108
`librt.a`	79 368	29 406
`libutil.a`	11 464	9 188
TOTAL	5 068 940	2 306 324

Table 6: Static Library Sizes (glibc vs. uClibc)

Shared Object	**glibc [bytes]**	**uClibc [bytes]**
`libc.so`	1 673 805	717 176
`ld.so`	148 652	35 856
`libcrypt.so`	28 748	13 676
`libdl.so`	16 303	13 716
`libm.so`	563 876	80 040
`libnsl.so`	108 321	5 032
`libpthread.so`	120 825	97 189
`libresolv.so`	88 470	5 036
`librt.so`	45 042	14 468
`libutil.so`	13 432	9 320
TOTAL	2 807 474	991 509

Table 7: Shared Object Sizes (glibc vs. uClibc)

RESULT	glibc	uClibc
TOTAL	1830	1648
PASSED	1447	1373
FAILED	111	83
UNRESOLVED	151	95
UNSUPPORTED	22	29
UNTESTED	92	60
INTERRUPTED	0	0
HUNG	1	3
SEGV	5	5
OTHERS	1	0

Table 8: OPT Results (glibc vs. uClibc)

most likely been the same. The remaining results are still being analyzed and will be presented at the Ottawa Linux Symposium in July, 2006. The complete test logs are available from the uClibc NPTL project website.

7 Conclusions

NPTL support in uClibc is now a reality. A fully POSIX compliant threads library in uClibc is a great technology enabler for embedded systems developers who need fast multithreading capability in a small memory footprint. The results from the Open POSIX Testsuite need to be analyzed in greater detail in order to better quantify what, if any POSIX support is missing.

8 Future Work

There is still much work to be done for the uClibc NPTL implementation. Below is a list of the important items:

- Sync NPTL code in uClibc tree with latest glibc mainline code.
- Implement NPTL for other processor architectures.
- Get static libraries working for NPTL.
- Merge uClibc-NPTL branch with uClibc trunk.
- Implement POSIX message queues.
- Implement Asynchronous I/O.
- Implement `sigqueue` call.
- Fix outstanding LTP and Open POSIX Test failures.

9 Acknowledgements

I would first like to acknowledge and thank God for getting me through this project. It has been a 9-1/2 month journey full of difficulty and frustration at times. His strength kept me going. Secondly, my wife Jennifer who was a constant encourager and supporter of me. I spent many weekends and evenings working while she kept the household from falling apart. Obviously, I would like to thank Broadcom Corporation for supporting this development effort and providing the code back to the community. Thanks to Erik Andersen from Code Poet Consulting who was my partner in this endeavor, handling all the contracts and legal issues for me. Thank you to AMD for supplying me multiple MIPS development boards free of charge. Special thanks to Mathieu Chouinard for formatting my paper in the final hours before submittal. Finally, I would like to dedicate this paper and entire effort to my first child, Zachary James Hill who just turned a year old on March 5th, 2006. I love you son.

References

[1] GNU C Libary at `http://www.gnu.org/software/libc/` `http://sourceware.org/glibc/`

[2] uClibc at `http://www.uclibc.org/`

[3] Newlib at `http://sourceware.org/newlib/`

[4] Diet Libc at `http://www.fefe.de/dietlibc/`

[5] Klibc at `ftp://ftp.kernel.org/pub/linux/libs/klibc/`

[6] ELF Handling for Thread Local Storage at `http://people.redhat.com/drepper/tls.pdf`

[7] Binutils at `http://www.gnu.org/software/binutils/`

[8] GCC at `http://gcc.gnu.org/`

[9] NPTL Linux/MIPS at `http://www.linux-mips.org/wiki/NPTL`

[10] Hubertus Franke, Matthew Kirkwood, Rusty Russell. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*, pages 479–494, June 2002.

[11] Futexes Are Tricky at `http://people.redhat.com/drepper/futex.pdf`

[12] `pthread_cancel` function definition at `http://www.opengroup.org/onlinepubs/007908799/xsh/pthread_cancel.html`

[13] uClibc NPTL Project at `http://www.realitydiluted.com/nptl-uclibc/`

[14] The Native POSIX Thread Library for Linux at `http://people.redhat.com/drepper/nptl-design.pdf`

[15] Linux Test Project at `http://ltp.sourceforge.net/`

[16] Open POSIX Test Suite at `http://posixtest.sourceforge.net/`

[17] buildroot at `http://buildroot.uclibc.org/`

[18] crosstool at `http://www.kegel.com/crosstool/`

Playing BlueZ on the D-Bus

Marcel Holtmann
BlueZ Project
marcel@holtmann.org

Abstract

The integration of the Bluetooth technology into the Linux kernel and the major Linux distributions has progressed really fast over the last two years. The technology is present almost everywhere. All modern notebooks and mobile phones are shipped with built-in Bluetooth. The use of Bluetooth with a Linux based system is easy and in most cases it only needs an one-time setup, but all the tools are still command line based. In general this is not so bad, but for a greater success it is needed to seamlessly integrate the Bluetooth technology into the desktop. There have been approaches for the GNOME and KDE desktops. Both have been quite successful and made the use of Bluetooth easy. The problem however is that both implemented their own framework around the Bluetooth library and its daemons and there were no possibilities for programs from one system to talk to the other. With the final version of the D-Bus framework and its adaption into the Bluetooth subsystem of Linux, it will be simple to make all applications Bluetooth aware.

The idea is to establish one central Bluetooth daemon that takes care of all task that can't or shouldn't be handled inside the Linux kernel. These jobs include PIN code and link key management for the authentication and encryption, caching of device names and services and also central control of the Bluetooth hardware. All possible tasks and configuration options are accessed via the D-Bus interface. This will allow to abstract the internals of GNOME and KDE applications from any technical details of the Bluetooth specification. Even other application will get access to the Bluetooth technology without any hassle.

1 Introduction

The Bluetooth specification [1] defines a clear abstraction layer for accessing different Bluetooth hardware options. It is called the *Host Controller Interface* (HCI) and is the basis of all Bluetooth protocols stacks (see Figure 1).

This interface consists of commands and events that provide support for configuring the local device and creating connections to other Bluetooth devices. The commands are split into six different groups:

- Link Control Commands
- Link Policy Commands
- Host Controller and Baseband Commands
- Informational Parameters
- Status Parameters
- Testing Commands

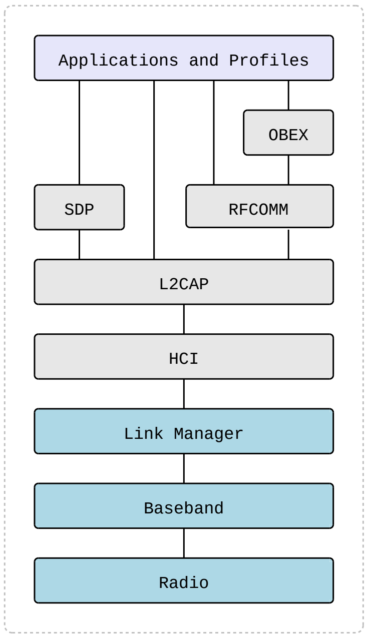


Figure 1: Simple Bluetooth stack

With the *Link Control Commands* it is possible to search for other Bluetooth devices in range and to establish connections to other devices. This group also includes commands to handle authentication and encryption. The *Link Policy Commands* are controlling the established connections between two or more Bluetooth devices. They also control the different power modes. All local settings of a Bluetooth device are modified with commands from the *Host Controller and Baseband Commands* group. This includes for example the friendly name and the class of device. For detailed information of the local device, the commands from the *Informational Paramters* group can be used. The *Status Parameters* group provides commands for detailed information from the remote device. This includes the link quality and the RSSI value. With the group *Testing Commands* the device provides commands for Bluetooth qualification testing. All commands are answered by an event that returns the requested value or information. Some events can also arrive at any time. For example to request a PIN code or to notify of a changed power state.

Every Bluetooth implementation must implement the *Host Controller Interface* and for Linux a specific set of commands has been integrated into the Linux kernel. Another set of commands are implemented through the Bluetooth library. And some of the commands are not implemented at all. This is because they are not needed or because they have been deprecated by the latest Bluetooth specification. The range of commands implemented in the kernel are mostly dealing with Bluetooth connection handling. The commands in the Bluetooth library are for configuration of the local device and handling of authentication and encryption.

While the *Host Controller Interface* is a clean hardware abstraction, it is not a clean or easy programming interface. The Bluetooth library provides an interface to HCI and an application programmer has to write a lot of code to get Bluetooth specific tasks done via HCI. To make it easy for application programmers and also end users, a task based interface to Bluetooth has been designed. The definition of this tasks has been done from an application perspective and they are exported through D-Bus via methods and signal.

2 D-Bus integration

The `hcid` daemon is the main daemon when running Bluetooth on Linux. It handles all device configuration and authentication tasks. All configuration is done via a simple configuration file and the PIN code is handled via PIN helper script. This means that every the configuration option needed to be changed, it was needed to edit the configuration file (`/etc/bluetooth/hcid.conf`) and to restart `hcid`. The configuration file still configures the basic and also default settings of `hcid`, but with the D-Bus integration all other settings are configurable through the D-Bus API. The current API consists of three interfaces:

- org.bluez.Manager
- org.bluez.Adapter
- org.bluez.Security

The *Manager* interface provides basic methods for listing all attached adapter and getting the default adapter. In the D-Bus API terms an adapter is the local Bluetooth device. In most cases this might be an USB dongle or a PCMCIA card. The *Adapter* interface provides methods for configuration of the local device, searching for remote device and handling of remote devices. The *Security* interface provides methods to register passkey agents. These agents can provide fixed PIN codes, dialog boxes or wizards for specific remote devices. All Bluetooth applications using the D-Bus API don't have to worry about any Bluetooth specific details or details of the Linux specific implementation (see Figure 2).

Besides the provided methods, every interface contains also signals to broadcast changes or events from the HCI. This allows passive applications to get the information without actively interacting with any Bluetooth related task. An example for this would be an applet that changes its icon depending on if the local device is idle, connected or searching for other devices.

Every local device is identified by its path. For the first Bluetooth adapter, this would be `/org/bluez/hci0` and this path will be used for all methods of the *Adapter* interface. The best way to get this path is to call `DefaultAdapter()` from the *Manager* interface. This will always return the current default adapter or in error if no Bluetooth adapter is attached. With `ListAdapters()` it is possible to get a complete list of paths of the attached adapters.

If the path is known, it is possible to use the full *Adapter* interface to configure the local device or handle tasks like pairing or searching for other devices. An example task would be the configuration of the device name. With `GetName()` the current name can be retrieved and with `SetName()` it can be changed. Changing the name results in storing it on the filesystem and changing the name with an appropriate HCI command. If the local device already supports the Bluetooth Lisbon specification, then the *Extended Inquiry Response* will be also modified.

With the `DiscoverDevices()` method it is possible to start the search for other Bluetooth devices in range. This method call actually doesn't return any remote devices. It only starts the inquiry procedure of the Bluetooth chip and every found device is returned via the `RemoteDeviceFound` signal. This allows all applications to handle new devices even if the discovery procedure has been initiated by a different application.

3 Current status

The methods and signals for the D-Bus API for Bluetooth were chosen very carefully. The goal was to design it with current application needs in mind. It also aims to fulfill the needs of current established desktop frameworks like the *GNOME Bluetooth subsystem* and the *KDE Bluetooth framework*. So it covers the common tasks and on purpose not everything that might be possible. The API can be divided into the following sections:

- Local

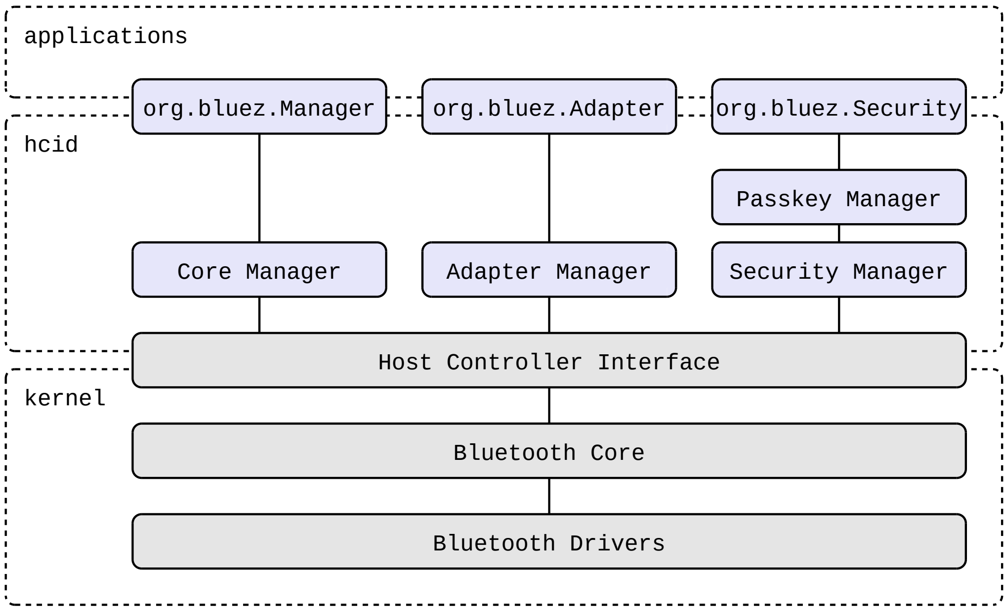


Figure 2: D-Bus API overview

  - version, revision, manufacturer
  - mode, name, class of device

- Remote
  - version, revision, manufacturer
  - name, class of device
  - aliases
  - device discovery
  - pairing, bondings

- Security
  - passkey agent

With these methods and signals all standard tasks are covered. The *Manager*, *Adapter* and *Security* interfaces are feature complete at the moment.

4 Example application

The big advantage of the D-Bus framework is that it has bindings for multiple programming languages. With the integration of D-Bus into the Bluetooth subsystem, the use of Bluetooth from various languages becomes reality. The Figure 3 shows an example of changing the name of the local device into *My Bluetooth dongle* using the Python programming language.

The example in Python is straight forward and simple. Using the D-Bus API within a C program is a little bit more complex, but it is still easier than using the native Bluetooth library API. Figure 4 shows an example on how to get the name of the local device.

5 Conclusion

The integration of a D-Bus API into the Bluetooth subsystem makes it easy for applications to access the Bluetooth technology. The current API is a big step into the right direction, but it is still limited. The Bluetooth technology is complex and Bluetooth services needs to be extended with an easy to use D-Bus API.

```
#!/usr/bin/python

import dbus

bus = dbus.SystemBus();

obj = bus.get_object('org.bluez',
            '/org/bluez')

manager = dbus.Interface(obj,
            'org.bluez.Manager')

obj = bus.get_object('org.bluez',
            manager.DefaultAdapter())

adapter = dbus.Interface(obj,
            'org.bluez.Adapter')

adapter.SetName('My Bluetooth dongle')
```

Figure 3: Example in Python

```
#include <stdio.h>
#include <stdlib.h>

#include <dbus/dbus.h>

int main(int argc, char **argv) {
 DBusConnection *conn;
 DBusMessage *msg, *reply;
 const char *name;

 conn = dbus_bus_get(DBUS_BUS_SYSTEM, NULL);
 msg  = dbus_message_new_method_call(
         "org.bluez",
         "/org/bluez/hci0",
         "org.bluez.Adapter", "GetName");

 reply =
   dbus_connection_send_with_reply_and_block(
         conn, msg, -1, NULL);

 dbus_message_get_args(reply, NULL,
         DBUS_TYPE_STRING, &name,
         DBUS_TYPE_INVALID);

 printf("%s\n", name);

 dbus_message_unref(msg);
 dbus_message_unref(reply);
 dbus_connection_close(conn);

 return 0;
}
```

Figure 4: Example in C

The next steps would be integration of D-Bus into the Bluetooth mouse and keyboard service. Another goal is the seamless integration into the Network Manager. This would allow to connect to Bluetooth access points like any other WiFi access point.

The current version of the D-Bus API for Bluetooth will be used in the next generation of the *Maemo platform* which is that basis for the *Nokia 770* Internet tablet.

References

[1] Special Interest Group Bluetooth: *Bluetooth Core Specification Version 2.0 + EDR*, November 2004.

[2] freedesktop.org: *D-BUS Specification Version 0.11*.

FS-Cache: A Network Filesystem Caching Facility

David Howells
Red Hat UK Ltd
dhowells@redhat.com

Abstract

FS-Cache is a kernel facility by which a network filesystem or other service can cache data locally, trading disk space to gain performance improvements for access to slow networks and media. It can be used by any filesystem that wishes to use it, for example AFS, NFS, CIFS, and ISOFS. It can support a variety of backends: different types of cache that have different trade-offs.

FS-Cache is designed to impose as little overhead and as few restrictions as possible on the client network filesystem using it, whilst still providing the essential services.

The presence of a cache indirectly improves performance of the network and the server by reducing the need to go to the network.

1 Overview

The *FS-Cache* facility is intended for use with network filesystems, permitting them to use persistent local storage to cache data and metadata, but it may also be used to cache other sorts of media such as CDs.

The basic principle is that some media are effectively slower than others—either because they are physically slower, or because they must be shared—and so a cache on a faster medium can be used to improve general performance by reducing the amount of traffic to or across the slower media.

Another reason for using a cache is that the slower media may be unreliable for some reason—for example a laptop might lose contact with a wireless network, but the working files might still need to be available. A cache can help with this by storing the working set of data and thus permitting disconnected operation (offline working).

1.1 Organisation

FS-Cache is a thin layer (see Figure 1) in the kernel that permits client filesystems (such as *NFS*, *AFS*, *CIFS*, *ISOFS*) on one side to request caching services without knowing what sort of cache is attached, if any.

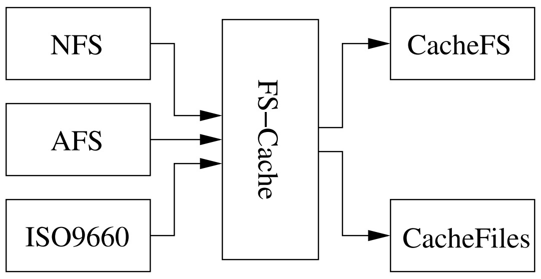


Figure 1: Cache architecture

On the other side *FS-Cache* farms those requests off to the available caches, be they *CacheFS*, *CacheFiles*, or whatever (see section 4)—or the request is gracefully denied if there isn't an available cache.

FS-Cache permits caches to be shared between several different sorts of *netfs*[1], though it does not in any way associate two different views of the same file obtained by two separate means. If a file is read by both *NFS* and *CIFS*, for instance, *two* copies of the file will end up in the cache (see section 1.7).

It is possible to have more than one cache available at one time. In such a case, the available caches have unique tags assigned to them, and a *netfs* may use these to bind a mount to a specific cache.

1.2 Operating Principles

FS-Cache does not itself require that a *netfs* file be completely loaded into the cache before that file may be accessed through the cache. This is because:

1. it must be practical to operate *without* a cache;
2. it must be possible to open a remote file that's *larger* than the cache;
3. the combined size of all open remote files—including mapped libraries—must not be limited to the size of the cache; and
4. the user should not be forced to download an entire file just to do a one-off access of a small portion of it (such as might be done with the `file` program).

[1]Note that the client filesystems will be referred to generically as the *netfs* in this document.

FS-Cache makes no use of the `i_mapping` pointer on the *netfs* inode as this would force the filesystems using the cache either to be bimodal[2] in implementation or to always require a cache for operation, with the files completely downloaded before use—none of which is acceptable for filesystems such as *NFS*.

FS-Cache is built instead around the idea that data should be served out of the cache in pages as and when requested by the *netfs* using it. That said, the *netfs* may, if it chooses, download the whole file and install it in the cache before permitting the file to be used—rejecting the file if it won't fit. All *FS-Cache* would see is a reservation (see section 1.4) followed by a stream of pages to entirely fill out that reservation.

Furthermore, *FS-Cache* is built around the principle that the *netfs*'s pages should belong to the *netfs*'s inodes, and so *FS-Cache* reads and writes data directly to or from those pages.

Lastly, files in the cache are accessed by sequences of keys, where keys are arbitrary blobs of binary data. Each key in a sequence is used to perform a lookup in an index to find the next index to consult or, finally, the file to access.

1.3 Facilities Provided

FS-Cache provides the following facilities:

1. More than one cache can be used at once. Caches can be selected explicitly by use of tags.
2. Caches can be added or removed at any time.

[2]Bimodality would involve having the filesystem operate very differently in each case

3. The *netfs* is provided with an interface that allows either party to withdraw caching facilities from a file (required for point 2). See section 5.

4. The interface to the *netfs* returns as few errors as possible, preferring rather to let the *netfs* remain oblivious. This includes I/O errors within the cache, which are hidden from the *netfs*. See section 5.8.

5. Cookies are used to represent indices, data files and other objects to the *netfs*. See sections 3 and 5.1.

6. Cache absence is handled gracefully; the *netfs* doesn't really need to do anything as the *FS-Cache* functions will just observe a NULL pointer—a negative cookie—and return immediately. See section 5.2.

7. Cache objects can be "retired" upon release. If an object is retired, *FS-Cache* will mark it as obsolete, and the cache backend will delete the object — data and all—and recursively retire all that object's children. See section 5.5.

8. The *netfs* is allowed to propose—dynamically—any index hierarchy it desires, though it must be aware that the index search function is recursive, stack space is limited, and indices can only be children of other indices. See section 3.2.

9. Data I/O is done on a page-by-page basis. Only pages which have been stored in the cache may be retrieved. Unstored pages are passed back to the *netfs* for retrieval from the server. See section 5.7.

10. Data I/O is done directly to and from the *netfs*'s pages. The *netfs* indicates that page A is at index B of the data-file represented by cookie C, and that it should be read or written. The cache backend may or may not start I/O on that page, but if it does, a *netfs* callback will be invoked to indicate completion. The I/O may be either synchronous or asynchronous.

11. A small piece of auxiliary data may be stored with each object. The format and usage of this data is entirely up to the *netfs*. The main purpose is for coherency management.

12. The *netfs* provides a "match" function for index searches. In addition to saying whether or not a match was made, this can also specify that an entry should be updated or deleted. This should make use of auxiliary data to maintain coherency. See section 5.4.

1.4 Disconnected Operation

Disconnected operation (offline working) requires that the set of files required for operation is fully loaded into the cache, so that the *netfs* can provide their contents without having to resort to the network. Not only that, it must be possible for the netfs to save changes into the cache and keep track of them for later synchronisation with the server when the network is once again available.

FS-Cache does not, of itself, provide disconnected operation. That facility is left up to the *netfs* to implement—in particular with regard to synchronisation of modifications with the server.

That said, *FS-Cache does* provide three facilities to make the implementation of such a facility possible: **reservations**, **pinning** and **auxiliary data**.

Reservations permit the *netfs* to reserve a chunk of the cache for a file, so that file can be loaded or expanded up to the specified limit.

Pinning permits the *netfs* to prevent a file from being discarded to make room in the cache for other files. The offline working set must be pinned in the cache to make sure it *will* be there when it's needed. The *netfs* would have to provide a way for the user to nominate the files to be saved, since they, and not the *netfs*, know what their working set will be.

Auxiliary data permits the *netfs* to keep track of a certain amount of writeback control information in the cache. The amount of primary auxiliary data is limited, but more can be made available by adding child objects to a data object to hold the extra information.

To implement potential disconnected operation for a file, the *netfs* must download all the missing bits of a file and load them into the cache in advance of the network going away.

Disconnected operation could also be of use with regard to *ISOFS*: the contents of a CD or DVD could be loaded into the cache for later retrieval without the need for the disc to be in the drive.

1.5 File Attributes

Currently arbitrary file attributes (such as *extended attributes* or *ACLs*) can be retained in the cache in one of two ways: either they can be stored in the auxiliary data (which is restricted in size - see section 1.4) or they can be attached to objects as children of a special object type (see section 3).

Special objects are data objects of a type that isn't one of the two primary types (index and data). How special objects are used is at the discretion of the *netfs* that created it, but special objects behave otherwise exactly like data objects.

Optimisations may be provided later to permit cache file extended attributes to be used to cache file attributes - especially with the possibility of attribute sharing on some backing filesystems. This will improve the performance of attribute-heavy systems such as those that use SE Linux.

1.6 Performance Trade-Offs

The use of a local cache for remote filesystems requires some trade-offs be made in terms of client machine performance:

- **File lookup time**

  This will be INCREASED by checking the cache before resorting to the network and also by making a note of a looked-up object in the cache.
  This should be DECREASED by local caching of metadata.

- **File read time**

  This will be INCREASED by checking the cache before resorting to the network and by copying the data obtained back to the cache.
  This should be DECREASED by local caching of data as a local disk should be quicker to read.

- **File write time**

  This could be DECREASED by doing writeback caching using the disk.
  Write-through caching should be more or less neutral since it's possible to write to both the network and the disk at once.

- **File replacement time**

  This will be INCREASED by having to retire an object or tree of objects from the disk.

The performance of the network and the server are also affected, of course, since the use of a local cache should hopefully reduce network traffic by satisfying from local storage some of the requests that would have otherwise been committed to the network. This may to some extent counter the increases in file lookup time and file read time due to the drag of the cache.

1.7 Cache Aliasing

As previously mentioned, through the interaction of two different methods of retrieving a file (such as *NFS* and *CIFS*), it is possible to end up with two or more copies of a remote file stored locally. This is known as *cache aliasing*.

Cache aliasing is generally considered bad for a number of reasons: it requires extra resources to maintain multiple copies, the copies may become inconsistent, and the process of maintaining consistency may cause the data in the copies to bounce back and forth. It's generally up to the user to avoid cache aliasing in such a situation, though the *netfs* can help by keeping the number of aliases down.

The current *NFS* client can also suffer from cache aliasing with respect to itself. If two mounts are made of different directories on the same server, then two superblocks will be created, each with its own set of inodes. Yet some of the inodes may actually represent the same file on the server, and would thus be aliases. Ways to deal with this are being examined.

FS-Cache deals with the possibility of cache aliasing by refusing multiple acquisitions of the same object (be it an index object or a data object). It is left up to the *netfs* to multiplex objects.

1.8 Direct File Access

Files opened with `O_DIRECT` should not go through the cache. That is up to the *netfs* to implement, and *FS-Cache* shouldn't even see the direct I/O operations.

If a file is opened for direct file access when there's data for that file in the cache, the cache object representing that file should be retired and a new one not created until the file is no longer open for direct access.

1.9 System Administration

Use of the *FS-Cache* facility by a *netfs* does not require anything special on the part of the system administrator, unless the *netfs* designer wills it. For instance, the in-kernel *AFS* filesystem will use it automatically if it's there, whilst the *NFS* filesystem currently requires an extra mount option to be passed to enable caching on that particular mount.

Whilst the exact details are subject to change, it should not be a problem to use the cache with automounted filesystems as there should be no need to wrap the mount call or issue a post-mount enabler.

2 Other Caching Schemes

Some network filesystems that can be used on Linux already have their own caching facilities built into each individually, including *Coda* and *OpenAFS*. In addition, other operating systems have caching facilities, such as *Sun*'s *CacheFS*.

2.1 *Coda*

Coda[1] requires a cache. It fully downloads the target file as part of the open process and stores it in the cache. The *Coda* file operations then redirect the various I/O operations to the equivalents on the cache file, and `i_mapping` is used to handle `mmap()` on a *Coda* file (this is required as *Coda* inodes do not have their own pages). `i_mapping` is not required with FS-Cache as the cache does I/O directly to the *netfs*'s pages, and so `mmap()` can just use the *netfs* inode's pages as normal.

All the changes made to a *Coda* file are stored locally, and the entire file is written back when a file is either flushed on `close()` or `fsync()`.

All this means that *Coda* may not handle a set of files that won't fit in its cache, and *Coda* can't operate *without* a cache. On the other hand, once a file has been downloaded, it operates pretty much at normal disk-file speeds. But imagine running the `file` program on a file of 100MB in size... Probably all that is required is the first page, but *Coda* will download *all* of it—that's fine if the file is then going to be used; but if not, that's a lot of bandwidth wasted.

This does, however, make *Coda* good for doing disconnected operation: you're guaranteed to have to hand the entirety of any file you were working with.

And it does potentially make *Coda* bad at handling sparse files, since *Coda* must download the whole file, holes and all, unless the Coda server can be made to pass on information about the gaps in a file.

2.2 *OpenAFS*

OpenAFS[2] can operate without a cache. It downloads target files piecemeal as the appropriate bits of the file are accessed, and places the bits in the cache if there is one.

No use is made of `i_mapping`, but instead *OpenAFS* inodes own their own pages, and the contents are exchanged with pages in the cache files at appropriate times.

OpenAFS's caching operates using the main model assumed for *FS-Cache*. *OpenAFS*, however, locates its cache files by invoking `iget()` on the cache superblock with the inode number for what it believes to be the cache file inode number as a parameter.

2.3 *Sun's CacheFS*

Modern *Solaris*[3] variants have their own filesystem caching facilities available for use with *NFS* (*CacheFS*). The mounting protocol is such that the cache must manually be attached to each *NFS* mount after the mount has been made.

FS-Cache does things a little differently: the *netfs* declares an interest in using caching facilities when the *netfs* is mounted, and the cache will be automatically attached either immediately if it's already available, or at the point it becomes available.

It would also be possible to get a *netfs* to request caching facilities after it has been mounted, though it might be trickier from an implementation point of view.

3 Objects and Indexing

Part of *FS-Cache* can be viewed as an **object** storage interface. The objects it stores come in two primary types: **index objects** and **data objects**, but other **special object** types may be defined on a per-parent-object basis as well.

Cache objects have certain properties:

- All objects apart from the root index object—which is inaccessible on the *netfs* side of things—have a parent object.
- Any object may have as many child objects as it likes.
- The children of an object do not all have to be of the same type.
- Index objects may only be the children of other index objects.
- Non-index objects[3] may carry data as well as children.
- Non-index objects have a file size set beyond which pages may not be accessed.
- Index objects may not carry data.
- Each object has a key that is part of a keyspace associated with its parent object.
- Child keyspaces from two separate objects do not overlap—so two objects with equivalent binary blobs as their keys but with different parent objects are *different* objects.
- Each object may carry a small blob of *netfs*-specific auxiliary metadata that can be used to manage cache consistency and coherence.
- An object may be pinned in the cache, preventing it from being culled to make space.
- A non-index object may have space reserved in the cache for data, thus guaranteeing a minimum amount of page storage.

Note that special objects behave exactly like data objects, except in two cases: when they're being looked up, the type forms part of the key; and when the cache is being culled, special objects are not automatically culled, but they are still removed upon request or when their parent object goes away.

[3]Data objects and special objects

3.1 Indices

Index objects are very restricted objects as they may only be the children of other indices and they may not carry data. However, they may exist in more than one cache if they don't have any non-index children, and they may be bound to specific caches—which binds all their children to the same cache.

Index object instantiation within any particular cache is deferred until an index further down the branch needs a non-index type child object instantiating within that cache—at which point the full path will be instantiated in one go, right up to the root index if necessary.

Indices are used to speed up file lookup by splitting up the key to a file into a sequence of logical sections, and can also be used to cut down keys that are too long to use in one lump. Indices may also be used define a logical group of objects so that the whole group can be invalidated in one go.

Records for index objects are created in the virtual index tree in memory whether or not a cache is available, so that cache binding information can be stored for when a cache is finally made available.

3.2 Virtual Indexing Tree

FS-Cache maintains a virtual indexing tree in memory for all the active objects it knows about. There's an index object at the root of the tree for *FS-Cache*'s own use. This is the **root index**.

The children of the root index are keyed on the name of the *netfs* that wishes to use the offered caching services. When a *netfs* requests caching services an index object specific to that service will be created if one does not already exist (see Figure 2).

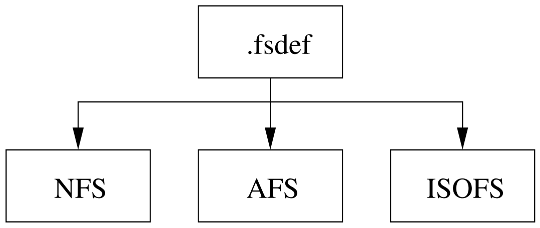


Figure 2: Primary Indices

Each of these is the **primary index** for the named *netfs* , and each can be used by its owner *netfs* in any way it desires. *AFS*, for example, would store per-cell indices in its primary index, using the cell name as the key.

Each primary index is versioned. Should a *netfs* request a primary index of a version other than the one stored in the cache, the entire index subtree rooted at that primary index will be scrapped, and a new primary index will be made.

Note that the index hierarchy maintained by a *netfs* will **not** normally reflect the directory tree that that *netfs* will display to the VFS and the user. Data objects generally are equivalent to inodes, not directory entries, and so hardlink and rename maintenance is not normally a problem for the cache.

For instance, with NFS the primary index might be used to hold an index per server—keyed by IP address—and each server index used to hold a data object per inode—keyed by NFS filehandle (see Figure 3).

The inode objects could then have child objects of their own to represent extended attributes or directory entries (see Figure 4).

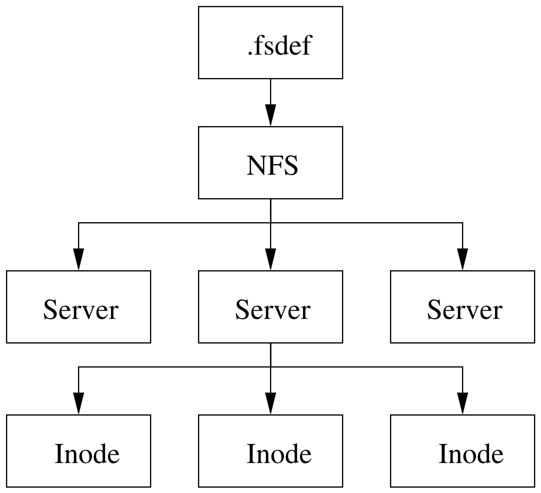


Figure 3: NFS Index Tree

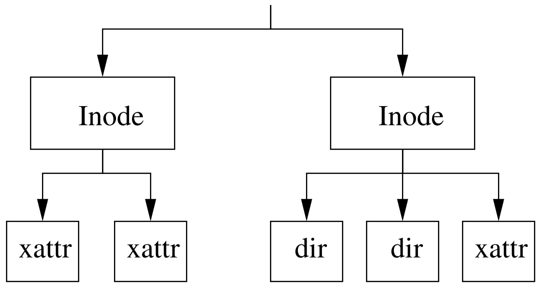


Figure 4: NFS Inode Attributes

Note that the in-memory index hierarchy may not be fully representative of the union of the on-disk trees in all the active caches on a system. *FS-Cache* may discard inactive objects from memory at any time.

3.3 Data-Containing Objects

Any data object may contain quantities of pages of data. These pages are held on behalf of the *netfs*. The pages are accessed by index number rather than by file position, and the object can be viewed as having a sparse array of pages attached to it.

Holes in this array are considered to represent pages as yet unfetched from the *netfs* server, and if *FS-Cache* is asked to retrieve one of these, it will return an appropriate error rather than just returning a block full of zeros.

Special objects may also contain data in exactly the same was as data objects can.

4 Cache Backends

The job of actually storing and retrieving data is the job of a **cache backend**. *FS-Cache* passes the requests from the *netfs* to the appropriate cache backend to actually deal with it.

There are currently two candidate cache backends:

- CacheFS
- CacheFiles

CacheFS is a quasi-filesystem that permits a block device to be mounted and used as a cache. It uses the mount system call to make the cache available, and so doesn't require any special activation interface. The cache can be deactivated simply by unmounting it.

CacheFiles is a cache rooted in a directory in an already mounted filesystem. This is more use where an extra block device is hard to come by, or re-partitioning is undesirable. This uses the VFS/VM filesystem interfaces to get another filesystem (such as *Ext3*) to do the requisite I/O on its behalf.

Both of these are subject to change in the future in their implementation details, and neither are fully complete at the time of writing this paper. See section 6 for information on the state of these components, and section 6.1 for performance data at the time of writing.

5 The *Netfs* Kernel Interface

The *netfs* kernel interface is documented in:

```
Documentation/filesystems/
caching/netfs-api.txt
```

The in-kernel client support can be obtained by including:

```
linux/fscache.h
```

5.1 Cookies

The *netfs* and *FS-Cache* talk to each other by means of **cookies**. These are elements of the virtual indexing tree that *FS-Cache* maintains, but they appear as opaque pointers to the *netfs*. They are of type:

```
struct fscache_cookie *
```

A `NULL` pointer is considered to be a negative cookie and represents an uncached object.

A *netfs* receives a cookie from *FS-Cache* when it registers. This cookie represents the primary index of this *netfs*. A *netfs* can acquire further cookies by asking *FS-Cache* to perform a lookup in an object represented by a cookie it already has.

When a cookie is acquired by a *netfs*, an object definition must be supplied. Object definitions are described using the following structure:

```
struct fscache_object_def
```

This contains the cookie name; the object type; and operations to retrieve the object key and auxiliary data, to validate an object read from disk by it auxiliary data, to select a cache, and to manage *netfs* pages.

Note that a *netfs*'s primary index is defined by *FS-Cache*, and is not subject to change.

5.2 Negative Cookies

A **negative cookie** is a `NULL` cookie pointer. Negative cookies can be used anywhere that non-negative cookies can, but with the effect that the *FS-Cache* header file wrapper functions return an appropriate error as fast as possible.

Note that attempting to acquire a new cookie from a negative cookie will simply result in another negative cookie. Attempting to store or retrieve a page using a negative cookie as the object specifier will simply result in `ENOBUFS` being issued.

FS-Cache will also issue a negative cookie if an error such as `ENOMEM` or `EIO` occurred, a non-index object's parent has no backing cache, the backing cache is being withdrawn from the system, or the backing cache is stopped due to an earlier fatal error.

5.3 Registering The *Netfs*

Before the *netfs* may access any of the caching facilities, it must register itself by calling:

```
fscache_register_netfs()
```

This is passed a pointer to the *netfs* definition.

The *netfs* definition doesn't contain a lot at the moment: just the *netfs*'s name and index structure version number, and a pointer to a table of per-*netfs* operations which is currently empty.

After a successful registration, the primary index pointer in the *netfs* definition will have been filled in with a pointer to the primary index object of the *netfs*.

The registration will fail if it runs out of memory or if there's another *netfs* of the same name already registered.

When a *netfs* has finished with the caching facilities, it should unregister itself by calling:

```
fscache_unregister_netfs()
```

This is also passed a pointer to the *netfs* definition. It will relinquish the primary index cookie automatically.

5.4 Acquiring Cookies

A *netfs* can acquire further cookies by passing a cookie it already has along with an object definition and a private datum to:

```
fscache_acquire_cookie()
```

The cookie passed in represents the object that will be the parent of the new one.

The private datum will be recorded in the cookie (if one is returned) and passed to the various callback operations listed in the object definition.

The cache will invoke those operations in the cookie definition to retrieve the key and the auxiliary data, and to validate the auxiliary data associated with an object stored on disk.

If the object requested is of non-index type, this function will search the cache to which the parent object is bound to see if the object is already present. If a match is found, the owning *netfs* will be asked to validate the object. The validation routine may request that the object be used, updated or discarded.

If a match is not found, an object will be created if sufficient disk space and memory are available, otherwise a negative cookie will be returned.

If the parent object is not bound to a cache, then a negative cookie will be returned.

Cookies may not be acquired twice without being relinquished in between. A *netfs* must itself deal with potential cookie multiplexing and aliasing—such as might happen with multiple mounts off the same *NFS* server.

5.5 Relinquishing Cookies

When a *netfs* no longer needs the object attached to a cookie, it should relinquish the cookie:

`fscache_relinquish_cookie()`

When this is called, the caller may also indicate that they wish the object to be retired permanently—in which case the object and all its children, its children's children, etc. will be deleted from the cache.

Prior to relinquishing a cookie, a *netfs* must have uncached **all** the pages read or allocated to that cookie, and all the child objects acquired on that cookie must have been themselves relinquished.

The primary index should not be relinquished directly. This will be taken care of when the *netfs* definition is unregistered.

5.6 Control Operations

There are a number of *FS-Cache* operations that can be used to control the object attached to a cookie.

`fscache_set_i_size()`
This is used to set the maximum file size on a non-index object. Error `ENOBUFS` will be obtained if an attempt is made to access a page beyond this size. This is provided to allow the cache backend to optimise the on-disk cache to store an object of this size; it does not imply that any storage will be set aside.

`fscache_update_cookie()`
This can be used to demand that the auxiliary data attached to an object be updated from a *netfs*'s own records. The auxiliary data may also be updated at other times, but there's no guarantee of when.

`fscache_pin_cookie()`
`fscache_unpin_cookie()`
These can be used to request that an object be pinned in the cache it currently resides and to unpin a previously pinned cache.

`fscache_reserve_space()`
This can be used to reserve a certain amount of disk space in the cache for a data object to store data in. The reservation will be extended to include for any metadata required to store the reserved data. A reservation may be cancelled by reducing the reservation size to zero.

The pinning and reservation operations may both issue error `ENOBUFS` to indicate that an object is unbacked, and error `ENOSPC` to indicate that there's not enough disk space to set aside some for pinning and reservation.

Both reservation and pinning persist beyond the cookie being released unless the cookie or one of its ancestors in the tree is also retired.

5.7 Data Operations

There are a number of *FS-Cache* operations that can be used to store data in the object attached to a cookie and then to retrieve it again. Note that *FS-Cache* must be informed of the maximum data size of a non-index object before an attempt is made to access pages in that object.

`fscache_alloc_page()`
This is used to indicate to the cache that a *netfs* page will be committed to the cache

at some point, and that any previous contents may be discarded without being read.

`fscache_read_or_alloc_page()`
This is used to request the cache attempt to read the specified page from disk, and otherwise allocate space for it if not present as it will be fetched shortly from the server.

`fscache_read_or_alloc_pages()`
This is used to read or allocate several pages in one go. This is intended to be used from the `readpages` address space operation.

`fscache_write_page()`
This is used to store a netfs page to a previously read or allocated cache page.

`fscache_uncache_page()`
`fscache_uncache_pagevec()`
These are used to release the reference put on a cache page or a set of cache pages by a read or allocate operation.

The allocate, read, and write operations will issue error `ENOBUFS` if the cookie given is negative or if there's no space on disk in the cache to honour the operation. The read operation will issue error `ENODATA` if asked to retrieve data it doesn't have but that it can reserve space for.

The read and write operations may complete asynchronously, and will make use of the supplied callback in all cases where I/O is started to indicate to the *netfs* the success or failure of the operation. If a read operation failed on a page, then the *netfs* will need to go back to the server.

5.8 Error Handling

FS-Cache handles many errors as it can internally and never lets the *netfs* see them, preferring to translate them into negative cookies or `ENOBUFS` as appropriate to the context.

Out-of-memory errors are normally passed back to the *netfs* , which is then expected to deal with them appropriately, possibly by aborting the operation it was trying to do.

I/O errors in a cache are more complex to deal with. If an I/O error happens in a cache, then the cache will be stopped. No more cache transactions will take place, and all further attempts to do cache I/O will be gracefully failed.

If the I/O error happens during cookie acquisition, then a negative cookie will be returned, and all caching operations based on that cookie will simply give further negative cookies or `ENOBUFS`.

If the I/O error happens during the reading of pages from the cache, then if any pages as yet unprocessed will be returned to the caller if the fscache reader function is still in progress; and any pages already committed to the I/O process will either complete normally, or will have their callbacks invoked with an error indication. In the latter case, the *netfs* should fetch the page from the server again.

If the I/O error happens during the writing of pages to the cache, then either the fscache write will fail with `ENOBUFS` or the callback will be invoked with an error. In either case, it can be assumed that the page is not safely written into the cache.

5.9 Data Invalidation And Truncation

FS-Cache does not provide data invalidation and truncation operations per-se. Instead the object should be retired (by relinquishing it with the retirement option set) and acquired anew. Merely shrinking the maximum file size down is not sufficient, especially as representations of extended attributes and suchlike may not be expunged by truncation.

6 Current State

The *FS-Cache* facility and its associated cache backends and *netfs* interfaces are not, at the time of writing, upstream. They are under development at Red Hat at this time. The states of the individual components are as follows:

- *FS-Cache*
  At this time *FS-Cache* is stable. New features may be added, but none are planned.
- *CacheFS*
  CacheFS is currently stalled. Although the performance numbers obtained are initially good, after a cache has been used for a while read-back performance degrades badly due to fragmentation. There are ways planned to ameliorate this, but they require implementation.
- *CacheFiles*
  CacheFiles has been prototyped and is under development at the moment in preference to *CacheFS* as it doesn't require a separate block device to be made available, but can instead run on an already mounted filesystem. Currently only *Ext3* is being used with it.
- *NFS*
  The *NFS* interface is sufficiently complete to give read/write access through the cache. It does, however, suffer from local cache aliasing problems that need sorting out.
- *AFS*
  The *AFS* interfaces is complete as far as the in-kernel *AFS* filesystem is currently able to go. *AFS* does *not* suffer from cache aliasing locally, but the filesystem itself does not yet have write support.

6.1 Current Performance

The caches have been tested with *NFS* to get some idea of the performance. *CacheFiles* was benchmarked on *Ext3* with 1K and 4K block sizes and on also *CacheFS*. The two caches and the block device raw tests were run on the same partition on the client's disk.

The client test machine contains a pair of 200MHz PentiumPro CPUs, 128MB of memory, an Ethernet Pro 100 NIC, and a Fujitsu MPG3204AT 20GB 5400rpm hard disk drive running in MDMA2 mode.

The server machine contains an Athlon64-FX51 with 5GB of RAM, an Ethernet Pro 100 NIC, and a pair of RAID1'd WDC WD2000JD 7200rpm SATA hard disk drives running in UDMA6 mode.

The client is connected through a pair of 100Mbps switches to the server, and the NFS connection was NFS3 over TCP. Before doing each test the files on the server were pulled into the server's pagecache by copying them to `/dev/null`. Each test was run several times, rebooting the client between iterations. The lowest number for each case was taken.

Reading a 100MB file:

Cache	*CacheFiles*		*CacheFS*
state	1K Ext3	4K Ext3	
None	26s	26s	26s
Cold	44s	35s	27s
Warm	19s	14s	11s

Reading 100MB of raw data from the same block device used to host the caches can be done in 11s.

And reading a 200MB file:

Cache	*CacheFiles*		*CacheFS*
state	1K Ext3	4K Ext3	
None	46s	46s	46s
Cold	79s	62s	47s
Warm	37s	29s	23s

Reading 200MB of raw data from the same block device used to host the caches can be done in 22s.

As can be seen, a freshly prepared *CacheFS* gives excellent performance figures, but these numbers don't show the degradation over time for large files.

The performance of *CacheFiles* will degrade over time as the backing filesystem does, if it does—but *CacheFiles*'s biggest problem is that it currently has to bounce the data between the *netfs* pages and the backing filesystems's pages. This means it does a *lot* of page-sized memory to memory copies. It also has to use `bmap` to probe for holes when retrieving pages, something that can be improved by implementing hole detection in the backing filesystem.

The performance of *CacheFiles* could possibly be improved by using direct I/O as well—that way the backing filesystem really would read and write directly from/to the *netfs*'s pages. That would obviate the need for backing pages and would reduce the large memory copies.

Note that *CacheFiles* is still being implemented, so these numbers are very preliminary.

7 Further Information

There's a mailing list available for *FS-Cache* specific discussions:

`mailto:linux-cachefs@redhat.com`

Patches may be obtained from:

`http://people.redhat.com/~dhowells/cachefs/`

and:

`http://people.redhat.com/~steved/cachefs/`

The FS-Cache patches add documentation into the kernel sources here:

`Documentation/filesystems/caching/`

References

[1] Information about *Coda* can be found at:
`http://www.coda.cs.cmu.edu/`

[2] Information about *OpenAFS* can be found at:
`http://www.openafs.org/`

[3] Information about *Sun*'s *CacheFS* facility can be found in their online documentation:
`http://docs.sun.com/`
Solaris 9 12/02 System Administrator Collection » System Administration Guide: Basic Administration » Chapter 40 Using The CacheFS File System (Tasks)
`http://docs.sun.com/app/docs/doc/816-4552/6maoo3121?a=view`

Why Userspace Sucks—Or 101 Really Dumb Things Your App Shouldn’t Do

Dave Jones
Red Hat
`<davej@redhat.com>`

Abstract

During the development of Fedora Core 5 I found myself asking the same questions day after day:

- Why does it take so long to boot?
- Why does it take so long to start X?
- Why do I get the opportunity to go fetch a drink after starting various applications and waiting for them to load?
- Why does idling at the desktop draw so much power?
- Why does it take longer to shut down than it does to boot up?

I initially set out to discover if there was something the kernel could do better to speed up booting. A number of suggestions have been made in the past ranging from better read-ahead, to improved VM caching strategies, or better on-disk block layout. I did not get that far however, because what I found in my initial profiling was disturbing.

We have an enormous problem with applications doing unnecessary work, causing wasted time, and more power-drain than necessary. This talk will cover a number of examples of common applications doing incredibly wasteful things, and will also detail what can be, and what has been done to improve the situation.

I intend to show by example numerous applications doing incredibly dumb things, from silliness such as reloading and re-parsing XML files 50 times each run, to applications that wake up every few seconds to ask the kernel to change the value of something that has not changed since it last woke up.

I created my tests using patches [1] to the Linux kernel, but experimented with other approaches using available tools like strace and systemtap. I will briefly discuss the use of these tools, as they apply to the provided examples, in later sections of this paper.

Our userspace sucks. Only through better education of developers of “really dumb things not to do” can we expect to resolve these issues.

1 Overview

A large number of strange things are happening behind the scenes in a lot of userspace programs. When their authors are quizzed about these discoveries, the responses range from “I had no idea it was doing that,” to “It didn’t do that on my machine." This paper hopes to address the former by shedding light on several tools (some old, some new) that enable userspace programmers to gain some insight into what is really going on. It addresses the latter by means of showing examples that may shock, scare, and embarrass their authors into writing code with better thought out algorithms.

2 Learning from read-ahead

Improving boot up time has been a targeted goal of many distributions in recent years, with each vendor resorting to a multitude of different tricks in order to shave off a few more seconds between boot and login. One such trick employed by Fedora is the use of a read-ahead tool, which, given a list of files, simply reads them into the page cache, and then exits. During the boot process there are periods of time when the system is blocked on some non-disk I/O event such as waiting for a DHCP lease. Read-ahead uses this time to read in files that are used further along in the boot process. By seeding the page cache, the start-up of subsequent boot services will take less time provided that there is sufficient memory to prevent it from being purged by other programs starting up during the time between the read-ahead application preloaded it, and the real consumer of the data starting up.

The read-ahead approach is a primitive solution, but it works. By amortising the cost of disk IO during otherwise idle periods, we shave off a significant amount of time during boot/login. The bootchart [2] project produced a number of graphs that helped visualise progress during the early development of this tool, and later went on to provide a rewritten version for Fedora Core 5 which improved on the bootup performance even further.

The only remaining questions are, what files do we want to prefetch, and how do we generate a list of them? When the read-ahead service was first added to Fedora, the file list was created using a kernel patch that simply printk'd the filename of every file open()'d during the first five minutes of uptime. (It was necessary to capture the results over a serial console, due to the huge volume of data overflowing the dmesg ring buffer very quickly.)

This patch had an additional use however, which was to get some idea of just what IO patterns userspace was creating.

During Fedora Core 5 development, I decided to investigate these patterns. The hope was that instead of the usual approach of 'how do we make the IO scheduler better', we could make userspace be more intelligent about the sort of IO patterns it creates.

I started by extending the kernel patch to log all file IO, not just open()s. With this new patch, the kernel reports every stat(), delete(), and path_lookup(), too.

The results were mind-boggling.

- During boot-up, 79576 files were stat()'d. 26769 were open()'d, 1382 commands were exec'd.
- During shutdown, 23246 files were stat()'d, 8724 files were open()'d.

2.1 Results from profiling

Picking through a 155234 line log took some time, but some of the things found were truly spectacular.

Some of the highlights included:

- HAL Daemon.

  - Reread and reparsed *dozens* of XML files during startup. (In some cases, it did this 54 times per XML file).
  - Read a bunch of files for devices that were not even present.
  - Accounted for a total of 1918 open()'s, and 7106 stat()'s.

- CUPS
  - Read in ppd files describing every printer known to man. (Even though there was not even a printer connected.)
  - Responsible for around 2500 stat()'s, and around 500 open()'s.

- Xorg

  A great example of how not to do PCI bus scanning.

  - Scans through /proc/bus/pci/ in order.
  - **Guesses** at random bus numbers, and tries to open those devices in /proc/bus/pci/.
  - Sequentially probes for devices on busses 0xf6 through 0xfb (even though they may not exist).
  - Retries entries that it has already attempted to scan regardless of whether they succeeded or not.

  Aside from this, when it is not busy scanning non-existent PCI busses, X really likes to stat and reopen lot of files it has already opened, like libGLcore.so. A weakness of its dynamic loader perhaps?

- XFS
  - Was rebuilding the font cache every time it booted, even if no changes had occurred in the fonts directories.

- gdm / gnome-session.
  - Tried to open a bunch of non-existent files with odd-looking names like `/usr/share/pixmaps/Bluecurve/cursors/00000000000000000000000000`
  - Suffers from font madness (See below).

2.2 Desktop profiling

Going further, removing the "first 5 minutes" check of the patch allowed me to profile what was going on at an otherwise idle desktop.

- irqbalance.
  - Wakes up every 10 seconds to rebalance interrupts in a round-robin manner. Made a silly mistake where it was re-balancing interrupts where no IRQs had ever occurred. A three line change saved a few dozen syscalls.
  - Was also re-balancing interrupts where an IRQ had not occurred in some time every 10 seconds.
  - Did an open/write/close of each `/proc/irq/n/smp_affinity` file each time it rebalanced, instead of keeping the fd's open, and doing 1/3rd of syscalls.

  Whilst working with /proc files does not incur any I/O, it does trigger a transition to-and-from kernel space for each system call, adding up to a lot of unneeded work on an otherwise 'idle' system.

- gamin
  - Was stat()'ing a bunch of gnome menu files every few seconds for no apparent reason.

% time	seconds	usecs/call	calls	errors	syscall
32.98	0.003376	844	4		clone
27.87	0.002853	4	699	1	read
23.50	0.002405	32	76		getdents
10.88	0.001114	0	7288	10	stat
1.38	0.000141	0	292		munmap
1.31	0.000134	0	785	382	open

Figure 1: `strace -c` output of gnome-terminal with lots of fonts.

- nautilus
  - Was stat'ing `$HOME/Templates`, `/usr/share/applications`, and `$HOME/.local/share/applications` every few seconds even though they had not changed.
- More from the unexplained department...
  - mixer_applet2 did a real_lookup on libgstffmpegcolorspace.so for some bizarre reason.
  - Does trashapplet really need to stat the svg for every size icon when it is rarely resized?

2.3 Madness with fonts

I had noticed through reviewing the log, that a lot of applications were stat()'ing (and occasionally open()'ing) a bunch of fonts, and then never actually using them. To try to make problems stand out a little more, I copied 6000 TTF's to $HOME/.fonts, and reran the tests. The log file almost doubled in size.

Lots of bizarre things stood out.

- gnome-session stat()'d 2473 and open()'d 2434 ttfs.
- metacity open()'d another 238.
- Just to be on the safe side, wnck-applet open()'d another 349 too.
- Nautilus decided it does not want to be left out of the fun, and open()'d another 301.
- mixer_applet rounded things off by open()ing 860 ttfs.

gnome-terminal was another oddball. It open()'ed 764 fonts and stat()'d another 770 including re-stat()'ing many of them multiple times. The vast majority of those fonts were not in the system-wide fonts preferences, nor in gnome-terminals private preferences. `strace -c` shows that gnome-terminal spends a not-insignificant amount of its startup time, stat()'ing a bunch of fonts that it never uses. (See Figure 1.)

Another really useful tool for parsing huge strace logs is Morten Wellinders strace-account [3] which takes away a lot of the tedious parsing, and points out some obvious problem areas in a nice easy-to-read summary.

Whilst having thousands of fonts is a somewhat pathological case, it is not uncommon for users to install a few dozen (or in the case of arty types, a few hundred). The impact of this defect will be less for most users, but it is still doing a lot more work than it needs to.

After my initial experiments were over, Dan Berrange wrote a set of systemtap scripts [4] to provide similar functionality to my tracing kernel patch, without the need to actually patch and rebuild the kernel.

3 Learn to use tools at your disposal

Some other profiling techniques are not as intrusive as to require kernel modifications, yet remarkably, they remain under-utilised.

3.1 valgrind

For some unexplained reason, there are developers that still have not tried (or in many cases, have not heard of) valgrind [5]. This is evident from the number of applications that still output lots of scary warnings during runtime.

Valgrind can find several different types of problems, ranging from memory leaks, to the use of uninitialised memory. Figure 2 shows an example of mutt running under valgrind.

The use of uninitialised memory can be detected without valgrind, by setting the environment variable `_MALLOC_PERTURB_` [6] to a value that will cause glibc to poison memory allocated with malloc() to the value the variable is set to, without any need to recompile the program.

Since Fedora Core 5 development, I run with this flag set to %RANDOM in my .bashrc. It adds some overhead to some programs which call malloc() a lot, but it has also found a number of bugs in an assortment of packages. A gdb backtrace is usually sufficient to spot the area of code that the author intended to use a calloc() instead of a malloc(), or in some cases, had an incorrect memset call after the malloc returns.

3.2 oprofile

Perceived by many as complicated, oprofile is actually remarkably trivial to use. In a majority of cases, simply running

```
opcontrol --start
(do application to
 be profiled)
opcontrol --shutdown
opreport -l
```

is sufficient to discover the functions where time is being spent. Should you be using a distribution which strips symbols out to separate packages (for example, Fedora/RHEL's -debuginfos), you will need to install the relevant -debuginfo packages for the applications and libraries being profiled in order to get symbols attributed to the data collected.

3.3 Heed the warnings

A lot of developers ignore, or even suppress warnings emitted by the compiler, proclaiming "They are just warnings." On an average day, the Red Hat package-build system emits around 40–50,000 warnings as part of its daily use giving some idea of the scale of this problem.

Whilst many warnings are benign, there are several classes of warnings that can have undesirable effects. For example, an implicit declaration warning may still compile and run just fine on your 32-bit machine, but if the compiler assumes the undeclared function has int arguments when it actually has long arguments, unusual results may occur when the code is run on a 64-bit machine. Leaving warnings unfixed makes it easier for real problems to hide amongst the noise of the less important warnings.

```
==20900== Conditional jump or move depends on uninitialised value(s)
==20900==    at 0x3CDE59E76D: re_compile_fastmap_iter (in /lib64/libc-2.4.so)
==20900==    by 0x3CDE59EBFA: re_compile_fastmap (in /lib64/libc-2.4.so)
==20900==    by 0x3CDE5B1D23: regcomp (in /lib64/libc-2.4.so)
==20900==    by 0x40D978: ??? (color.c:511)
==20900==    by 0x40DF79: ??? (color.c:724)
==20900==    by 0x420C75: ??? (init.c:1335)
==20900==    by 0x420D8F: ??? (init.c:1253)
==20900==    by 0x422769: ??? (init.c:1941)
==20900==    by 0x42D631: ??? (main.c:608)
==20900==    by 0x3CDE51D083: __libc_start_main (in /lib64/libc-2.4.so)
```

Figure 2: Mutt under valgrind

Bonus warnings can be enabled with compiler options `-Wall` and `-Wextra` (this option used to be `-W` in older gcc releases)

For the truly anal, static analysis tools such as splint [7], and sparse [8] may turn up additional problems.

In March 2006, a security hole was found in Xorg [9] by the Coverity Prevent scanner [10]. The code looked like this.

```
if (getuid() == 0 || geteuid != 0)
```

No gcc warnings are emitted during this compilation, as it is valid code, yet it does completely the wrong thing. Splint on the other hand indicates that something is amiss here with the warning:

```
Operands of != have incompatible types
([function (void) returns
__uid_t], int): geteuid != 0
Types are incompatible.
(Use -type to inhibit warning)
```

Recent versions of gcc also allow programs to be compiled with the `-D_FORTIFY_SOURCE=2` which enables various security checks in various C library functions. If the size of memory passed to functions such as memcpy is known at compile time, warnings will be emitted if the len argument overruns the buffer being passed.

Additionally, use of certain functions without checking their return code will also result in a warning. Some 30-40 or so C runtime functions have had such checks added to them.

It also traps a far-too-common[1] bug: memset with size and value arguments transposed. Code that does this:

```
memset(ptr, sizeof(foo), 0);
```

now gets a compile time warning which looks like this:

```
warning:  memset used with
constant zero length parameter;
this could be due to transposed
parameters
```

Even the simpler (and deprecated) bzero function is not immune from screwups of the size parameter it seems, as this example shows:

```
bzero(pages + npagesmax, npagesmax
- npagesmax);
```

Another useful gcc feature that was deployed in Fedora Core 5 was the addition of a stack overflow detector. With all applications compiled with the flags

[1]Across 1282 packages in the Fedora tree, 50 of them had a variant of this bug.

```
-fstack-protector --param=
ssp-buffer-size=4
```

any attempt at overwriting an on-stack buffer results in the program being killed with the following message:

```
*** stack smashing detected ***:
./a.out terminated
Aborted (core dumped)
```

This turned up a number of problems during development, which were usually trivial to fix up.

4 Power measurement

The focus of power management has traditionally been aimed at mobile devices; lower power consumption leads to longer battery life. Over the past few years, we have seen increased interest in power management from data centers, too. There, lowering power consumption has a direct affect on the cost of power and cooling. The utility of power savings is not restricted to costs though, as it will positively affect up-time during power outages, too.

We did some research into power usage during Fedora Core 5 development, to find out exactly how good/bad a job we were doing at being idle. To this end, I bought a 'kill-a-watt' [11] device (and later borrowed a 'Watts-up' [12] which allowed serial logging). The results showed that a completely idle EM64T box (Dell Precision 470) sucked a whopping 153 Watts of power. At its peak, doing a kernel compile, it pulled 258W, over five times as much power as its LCD display. By comparison, a VIA C3 Nehemiah system pulled 48 Watts whilst idle. The lowest power usage I measured on modern hardware was 21W idle on a mobile-Athlon-based Compaq laptop.

Whilst vastly lower than the more heavyweight systems, it was still higher than I had anticipated, so I investigated further as to where the power was being used. For some time, people have been proclaiming the usefulness of the 'dynamic tick' patch for the kernel, which stops the kernel waking up at a regular interval to check if any timers have expired, instead idling until the next timer in the system expires.

Without the patch, the Athlon XP laptop idled at around 21W. With dynticks, after settling down for about a minute, the idle routine auto-calibrates itself and starts putting off delays. Suddenly, the power meter started registering…20,21,19,20,19,20,18,21,19,20,22 changing about once a second. Given the overall average power use went down below its regular idle power use, the patch does seem like a win. (For reference, Windows XP does not do any tricks similar to the results of the dynticks patch, and idles at 20W on the same hardware). Clearly the goal is to spend longer in the lower states, by not waking up so often.

Another useful side-effect of the dyntick patch was that it provides a /proc file that allows you to monitor which timers are firing, and their frequency. Watching this revealed a number of surprises. Figure 3 shows the output of this file (The actual output is slightly different, I munged it to include the symbol name of the timer function being called.)

- Kernel problems Whilst this paper focuses on userspace issues, for completeness, I will also enumerate the kernel issues that this profiling highlighted.
  - USB. Every 256ms, a timer was firing in the USB code. Apparently the USB 2.0 spec mandates this timer, but if there are no USB devices connected (as was the case when I measured), it does call into the question

```
peer_check_expire        181  crond
dst_run_gc               194  syslogd
rt_check_expire          251  auditd
process_timeout          334  hald
it_real_fn               410  automount
process_timeout          437  kjournald
process_timeout         1260
it_real_fn              1564  rpc.idmapd
commit_timeout          1574
wb_timer_fn             1615  init
process_timeout         1652  sendmail
process_timeout         1653
process_timeout         1833
neigh_periodic_timer    1931
process_timeout         2218  hald-addon-stor
process_timeout         3492  cpuspeed
delayed_work_timer_fn   4447
process_timeout         7620  watchdog/0
it_real_fn              7965  Xorg
process_timeout        13269  gdmgreeter
process_timeout        15607  python
cursor_timer_handler   34096
i8042_timer_func       35437
rh_timer_func          52912
```

Figure 3: `/proc/timertop`

what exactly it is doing. For the purposes of testing, I worked around this with a big hammer, and rmmod'd the USB drivers.

  - keyboard controller. At HZ/20, the i8042 code polls the keyboard controller to see if someone has hotplugged a keyboard/mouse or not.
  - Cursor blinking. Hilariously, at HZ/5 we wake up to blink the cursor. (Even if we are running X, and not sat at a VT)

- gdm
  - For some reason, gdm keeps getting scheduled to do work, even when it is not the active tty.
- Xorg
  - X is hitting `it_real_fn` a *lot* even if it is not the currently active VT. Ironically, this is due to X using its 'smart scheduler', which hits SIGALRM regularly, to punish X clients that are hogging the server. Running X with -dumbsched made this completely disappear. At the time it was implemented, itimer was considered the fastest way of getting a timer out of the kernel. With advances from recent years speeding up gettimeofday() through the use of vsyscalls, this may no longer be the most optimal way for it to go about things.
- python
  - The python process that kept waking up belonged to hpssd.py, a part of hplip. As I do not have a printer,

this was completely unnecessary.

By removing the unneeded services and kernel modules, power usage dropped another watt. Not a huge amount, but significant enough to be measured.

Work is continuing in this area for Fedora Core 6 development, including providing better tools to understand the huge amount of data available. Current gnome-power-manager CVS even has features to monitor `/proc/acpi` files over time to produce easy-to-parse graphs.

5 Conclusions.

The performance issues discussed in this paper are not typically reported by users. Or, if they are, the reports lack sufficient information to root-cause the problem. That is why it is important to continue to develop tools such as those outlined in this paper, and to run these tools against the code base moving forward. The work is far from over; rather, it is a continual effort that should be engaged in by all involved parties.

With increased interest in power management, not only for mobile devices, but for desktops and servers too, a lot more attention needs to be paid to applications to ensure they are "optimised for power." Further development of monitoring tools such as the current gnome-power-manager work is key to understanding where the problem areas are.

Whilst this paper pointed out a number of specific problems, the key message to be conveyed is that the underlying problem does not lie with any specific package. The problem is that developers need to be aware of the tools that are available, and be informed of the new tools being developed. It is through the use of these tools that we can make Linux not suck.

References

[1] `http://people.redhat.com/davej/filemon`

[2] `http://www.bootchart.org`

[3] `http://www.gnome.org/~mortenw/files/strace-account`

[4] `http://people.redhat.com/berrange/systemtap/bootprobe`

[5] `http://valgrind.org`

[6] `http://people.redhat.com/drepper/defprogramming.pdf`

[7] `http://www.splint.org`

[8] `http://www.codemonkey.org.uk/projects/git-snapshots/sparse`

[9] `http://lists.freedesktop.org/archives/xorg/2006-March/013992.html`
`http://blogs.sun.com/roller/page/alanc?entry=security_hole_in_xorg_6`

[10] `http://www.coverity.com`

[11] kill-a-watt: `http://www.thinkgeek.com/gadgets/electronic/7657`

[12] watts-up: `http://www.powermeterstore.com/plug/wattsup.php`