Linux 2.6 IO Performance Analysis, Quantification, and Optimization

Dominique A. Heger
DHTechnologies (DHT)
dheger@dhtusa.com

Richard Quinn
Richard Quinn Consulting
richardquinn72@gmail.com

1.0 Abstract

In any IT environment, it is a rather challenging task to accurately identify and resolve IO systems performance problems. Especially, when the systems support a wide range of workload patterns and the actual performance issues only surface under certain workload conditions.

This paper presents a novel taxonomy that characterizes in a structured and pragmatic manner the interrelationships and tradeoffs of the (rather complex) Linux 2.6 IO stack. The focus is on elaborating on the tools and techniques available in Linux 2.6 to analyze, quantify, and optimize workload-dependent IO performance. The argument made is that only a detailed, layered analysis of the Linux 2.6 application, file system, block IO layer, IO scheduler, and device driver IO chain allows optimizing the application workload onto the logical and physical IO resources. Further, the study proposes a new algorithm for the Linux 2.6 default IO scheduler CFQ. The algorithm is based on Hopfield Artificial Neural Networks (ANN) and addresses some of the potential queue and starvation issues found in the current CFQ implementation.

2.0 Introduction

The argument made throughout the study is that quantifying and optimizing the Linux 2.6 IO performance is a function of the actual IO path, the devices in the code path, as well as the workload presented to them (the actual workload driver). This study was initiated to discuss and document a rather simple, efficient, and effective methodology to quantify Linux 2.6 IO performance. Compared to other UNIX operating systems, Linux 2.6 rather deviates from an IO design perspective and provides the user community with a more layered IO stack. To illustrate, in Linux 2.6, the IO scheduler, as well as the read-ahead mechanism is an actual artifact of the OS, whereas in traditional UNIX systems, these components are incorporated into the filesystem framework. In this study, Section 3 elaborates on the current Linux 2.6 IO stack. Section 4 introduces the IO schedulers and discusses their respective performance behavior. Section 5 discusses the proposed IO evaluation and quantification methodology, and elaborates on the available Linux tools to quantify IO performance from the application layer down into the actual IO hardware. Section 6 introduces a new algorithm that focuses on optimizing the aggregate IO throughput behavior of the CFQ scheduler, and presents actual benchmark results utilizing the optimized IO scheduler. The study concludes in Section 7 by summarizing the accomplishments and discussing some future work items.

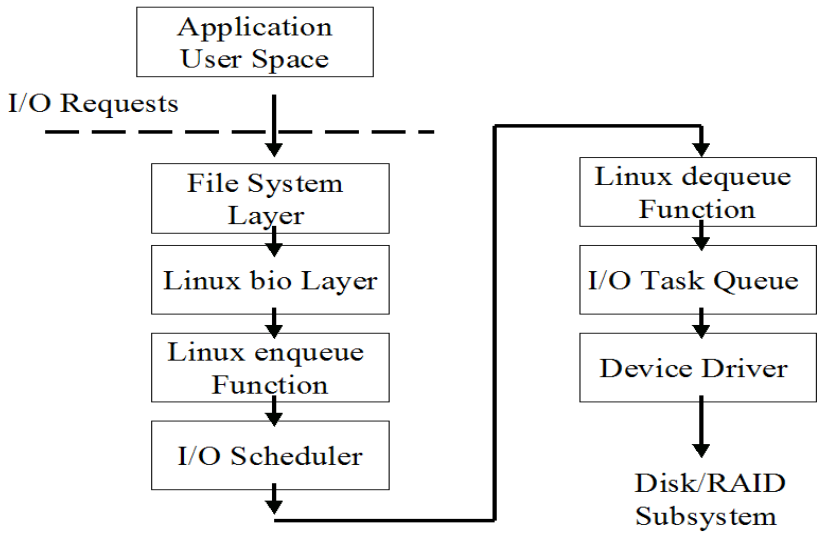
3.0 Linux 2.6 IO Framework

The I/O scheduler in Linux forms the interface between the generic block layer and the low-level device drivers. The block layer provides functions that are utilized by the file systems and the virtual memory manager to submit I/O requests to block layer (see Figure 1). These requests are transformed by the I/O scheduler and made available to the low-level device drivers. The device drivers consume the transformed requests and forward them (by using device specific protocols) to the actual device controllers that perform the I/O operations. As prioritized resource management seeks to regulate the use of a disk subsystem by an application, the I/O scheduler is considered an imperative kernel component in the Linux I/O path. It is further possible to regulate the disk usage in the kernel layers above and below the I/O scheduler. Adjusting the I/O pattern generated by the file system or the virtual memory manager (VMM) is considered as an option. Another option is to adjust the way specific device drivers or device controllers consume and manipulate the I/O requests.

The various Linux 2.6 I/O schedulers can be abstracted into a rather generic I/O model (Figure 1). The I/O requests are generated by the block layer on behalf of threads that are accessing various file systems, threads that are performing raw I/O, or are generated by virtual memory management (VMM) components of the kernel such as the *kswapd* or the *pdflush* threads. The producers of I/O requests initiate a call to *__make_request(),* which invokes various I/O scheduler functions such as *elevator_merge_fn()*. The *enqueue* functions in the I/O framework intend to merge the newly submitted block I/O unit (a *bio*) with previously submitted requests, and to sort (or sometimes just insert) the request into one or more internal

I/O queues. As a unit, the internal queues form a single logical queue that is associated with each block device [8]. At a later stage, the low-level device driver calls the generic kernel function *elv_next_request()* to obtain the next request from the logical queue. The *elv_next_request()* call interacts with the I/O scheduler's dequeue function *elevator_next_req_fn()*, and the latter has an opportunity to select the appropriate request from one of the internal queues. The device driver processes the request by converting the I/O submission into scatter-gather lists and protocol-specific commands that are submitted to the device controller [2]. From an I/O scheduler perspective, the block layer is considered as the producer of I/O requests and the device drivers are labeled as the actual consumers.

Figure 1: Linux 2.6 I/O Stack



From a generic IO perspective, every *read()* or *write()* request launched by an application results in either utilizing the respective I/O system calls, or in memory mapping (*mmap*) the file into a process's address space. I/O operations normally result in allocating *PAGE_SIZE* units of physical memory [4]. These pages are being indexed, as this enables the system to later on locate the page in the buffer cache. A cache subsystem though only improves performance if the data in the cache is being reused. Further, the read cache abstraction allows the system to implement read-ahead functionalities, as well as to construct large contiguous (SCSI or Fibre Channel) I/O commands that can be served via a single direct memory access (DMA) operation. In circumstances where the cache represents pure (memory bus) overhead, I/O features such as direct I/O should be explored (especially in situations where the system is CPU bound). In a general *write()* scenario, the system is not necessarily concerned with the previous content of a file, as a *write()* operation normally results in overwriting the contents in the first place. Therefore, the write cache emphasizes other aspects such as asynchronous updates, as well as the possibility of omitting some write requests in the case where multiple *write()* operations into the cache subsystem result in a single I/O operation to a physical IO component [9]. Such a scenario may occur in an environment where updates to the same (or a similar) inode offset are being processed within a rather short time-span. The representation of the block I/O layer in Linux 2.6 encourages large I/O operations. The block I/O layer tracks data buffers by using *struct page* pointers [2]. Linux 2.6 utilizes logical pages attached to inodes to flush dirty data, which allows multiple pages that belong to the same inode to be coalesced into a single *bio* that can be submitted to the I/O layer.

4.0 Linux 2.6 IO Schedulers

The next few paragraphs discuss the 4 IO schedulers available in the Linux 2.6 IO framework. Each IO scheduler has a different performance behavior, and hence, the internal working of each scheduler has to be known when designing and implementing Linux server systems. In this section, the focus is on the Linux 2.6 default IO scheduler CFQ. The noop, the anticipatory, and the deadline IO scheduler are discussed in this paper in a less comprehensive manner. Please see Appendix A for a detailed discussion of the Linux 3.x IO Scheduler framework (updated 2013).

4.1 CFQ IO Scheduler

The CFQ scheduler operates by placing synchronous requests that are submitted by threads into a number of per-thread queues, and then by allocating actual time-slices for each of the queues to access the (physical) IO subsystem. The length of the time-slice, as well as the number of IO requests that a queue is allowed to submit depends on the IO priority of the given thread. Asynchronous IO requests for all threads are bundled together (priority based) in a few queues. By design, good aggregate throughput behavior is achieved by allowing a thread queue to idle at the end of a synchronous IO batch, thereby allowing the IO framework to *anticipate* (short-term) close IO requests from the same thread (similar to the deadline IO scheduler discussed below). This behavior is considered a natural extension of granting actual IO time slices to a thread [8].

The designers behind the CFQ scheduler created the concept of having actual IO queues for each thread. These queues are created ad-hoc for each particular thread. Further, the designers segregated the concept of IO into two distinct sections, synchronous and asynchronous IO. Synchronous IO is important as the application threads have to stall until the IO request completes. To illustrate, if an application *read()* request does not hit in either the cache or the memory subsystem, the application *read()* request has to block until the data is available in memory, a scenario that obviously has a profound impact on application performance. In most circumstances, *write()* requests are asynchronous. Ergo, the *write()* requests are pushed into the memory subsystem, if possible consolidated, and flushed to disk either time-based, or based on the state of the memory subsystem.

Next to distinguishing between synchronous (which are favored) and asynchronous IO operations, the CFQ design also *prefers* read over write operations [3]. As already discussed in this paper, read requests have the potential to stall application processing, impacting aggregate systems performance. Further, read requests, based on the elevator approach, may starve other read requests that disclose (from an IO geometry perspective) long-distance cases (based on the currently processed IO batch). Therefore, favoring read over write operations (in some circumstances) may improve aggregate read IO responsiveness, and reduce the probability of read IO starvation. This problem is further addressed in this paper in Section 6. Based on the dynamic nature of IO operations (and the corresponding priority scenarios) though, there is always a possibility that an IO request gets pushed back in a queue, and hence gets delayed. To combat such a behavior, in CFQ, each thread is associated with a time-out value. If the IO thread is not executed within that epoch, as the time expires, the IO thread is instantly scheduled for execution.

As with all Linux 2.6 IO schedulers, the designers of the CFQ IO scheduler exported several tuning parameters into user space. In other words, the actual performance behavior of the CFQ scheduler, based on the application workload and the setup of the physical IO subsystem, can be adjusted/modified by the user community [8].

4.2 The noop IO Scheduler

The noop scheduler inserts the set of I/O requests into a simple, unordered FIFO queue and only provides simple IO request merging. The noop scheduler is very effective in environments where IO performance optimization is incorporated in a lower layer of the (physical) IO stack. In other words, the noop scheduler is normally chosen in cluster environments that have SAN access [9]. Further, the noop scheduler is best used with solid state disks (SSD) or any other device that can not benefit from re-ordering of multiple I/O requests at the Linux IO scheduler level.

4.3 The Anticipatory IO Scheduler

The anticipatory (AS) I/O scheduler's design attempts to reduce the per thread read response time. It introduces a controlled delay component into the dispatching equation. The delay is being invoked on any new *read()* request to the device driver, thereby allowing a thread that just finished its *read()* I/O request to submit a new *read()* request, basically enhancing the chances (based on locality) that this scheduling behavior will result in smaller seek operations. The tradeoff between reduced seeks and decreased disk utilization (due to the additional delay factor in dispatching a request) is managed by utilizing an actual *cost-benefit* analysis. More specifically, the Linux 2.6 implementation of the anticipatory I/O scheduler follows the basic idea that if the disk drive just operated on a *read()* request, the assumption can be made that there is another *read()* request in the pipeline, and hence it is worth the while to wait [5].

This concept basically addresses the *deceptive idleness* performance problem that may occur in other IO schedulers. Hence, the anticipatory I/O scheduler starts a timer, and at this point, there are no more I/O requests passed down to the device driver. If a (close) *read()* request arrives during the wait time, it is serviced immediately and in the process, the actual distance that the kernel considers as *close* grows as time passes (the adaptive part of the heuristic). Eventually the *close* requests will dry out and the scheduler will decide to submit some of the *write()* requests. Despite the fact that up until Linux 2.6.18, the anticipatory IO scheduler used to be the default scheduler for most of the Linux distributions, it is rarely being used today (some Web server systems may be the exception). Benchmarks have shown that the anticipatory IO scheduler may actually have a rather detrimental impact on IO performance with TCQ disk IO components or HW RAID systems, respectively.

4.4 The Deadline IO Scheduler

The deadline I/O scheduler incorporates a per-request expiration-based approach and operates on 5 I/O queues. The basic idea behind the implementation is to aggressively reorder requests to improve I/O performance while simultaneously ensuring that no I/O request is being starved [1]. More specifically, the scheduler introduces the notion of a per-request deadline, which is used to assign a higher preference to *read()* than *write()* requests. The scheduler maintains the 5 I/O queues. During the *enqueue* phase, each I/O request gets associated with a deadline, and is being inserted in I/O queues that are either organized by the starting logical block number (a sorted list) or by the deadline factor (a FIFO list). The scheduler incorporates separate sort and FIFO lists for *read()* and *write()* requests, respectively. The 5th I/O queue contains the requests that are to be handed off to the device driver. During a dequeue operation, in the case where the dispatch queue is empty, requests are moved from one of the 4 (sort or FIFO) I/O lists in batches.

The next step consists of passing the head request on the dispatch queue to the device driver (this scenario also holds true in the case that the dispatch-queue is not empty). The logic behind moving the I/O requests from either the sort or the FIFO lists is based on the scheduler's goal to ensure that each *read()* request is processed by its effective deadline, without starving the queued-up *write()* requests. In this design, the goal of economizing the disk seek time is accomplished by moving a larger batch of requests from the sort list (logical block number sorted), and balancing it with a controlled number of requests from the FIFO list. Hence, the ramification is that the deadline I/O scheduler effectively emphasizes average *read()* request response time over disk utilization and total average I/O request response time. Benchmarks have shown that some database applications do perform well with the deadline IO scheduler.

5.0 Methodology to Evaluate & Quantify IO Performance

Figure 2: Linux 2.6 strace example

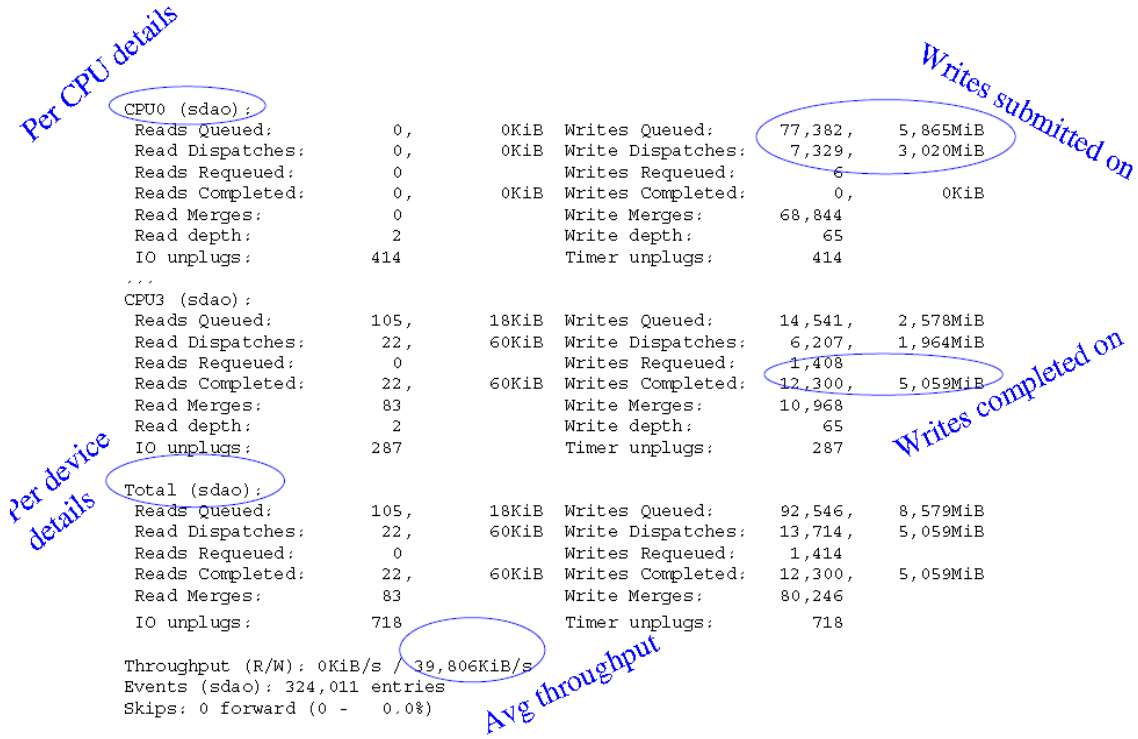
```
14:19:12.571300 open("/sys/block/sda//dev", O_RDONLY) = 5
14:19:12.571513 fstat64(5, {st_mode=S_IFREG|0444, st_size=4096, ...}) = 0
14:19:12.571759 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fb1000
14:19:12.571907 read(5, "8:0\n"..., 4096) = 4
14:19:12.572049 close(5)                = 0
14:19:12.572161 munmap(0xb7fb1000, 4096) = 0
14:19:12.572288 brk(0x9381000)          = 0x9381000
14:19:12.572403 close(4)                = 0
14:19:12.572517 open("/sys/block/sda/size", O_RDONLY) = 4
14:19:12.572766 fstat64(4, {st_mode=S_IFREG|0444, st_size=4096, ...}) = 0
14:19:12.572980 mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7fb1000
14:19:12.573161 read(4, "312581808\n"..., 4096) = 10
14:19:12.573347 close(4)                = 0
```

While quantifying systems IO performance, it is paramount to understand the workload distribution at the application layer first. The application is the actual workload driver, and its usage mix of read(), write(), mmap(), fstat(), or lseek() systems calls governs the IO throughput potential of any IO subsystem. An application can be data intensive, metadata intensive, operate in a mostly sequential or random IO framework, utilize small or large IO requests, or may be more read() or write() intensive. Tracing the actual application (in user space) allows determining the workload behavior, and establishing the IO workload profiles. The workload profiles, 1st explain the code path that the application is taking through the IO OS

stack, allowing for exact and precise tuning of the logical resources in the IO path. Establishing the IO path through the OS allows for precise systems tuning activities that are based on empirical data, verses the so common brute-force or *folklore* tuning approach (such as cut-and-paste out of a whitepaper) that is unfortunately so widespread in the IT community. 2nd, the workload profiles can be used as input in an IO benchmark (such as FFSB) to further quantify application IO performance under increased workload conditions (scalability analysis). In a Linux environment, the *strace* tool is being used to trace applications in user space [9]. No changes to the application code are required to trace the systems calls. The following *strace* example (Figure 2) highlights the potential of the Linux tool. In the above strace example, each line is time-stamped (down to the microsecond level). Hence, for each IO system call, the actual response time at the application level can be determined. Further, the IO request sizes are visible as arguments in the system calls. Therefore, for any individual IO call, the performance behavior at the application layer can be determined. A statistical analysis of the trace data allows determining if the application is read or write intensive, data or metadata intensive, or performs mainly sequential or random IO operations. Further, over the length of the trace, the total amount of data read or written, as well as the number of IO operations can be quantified.

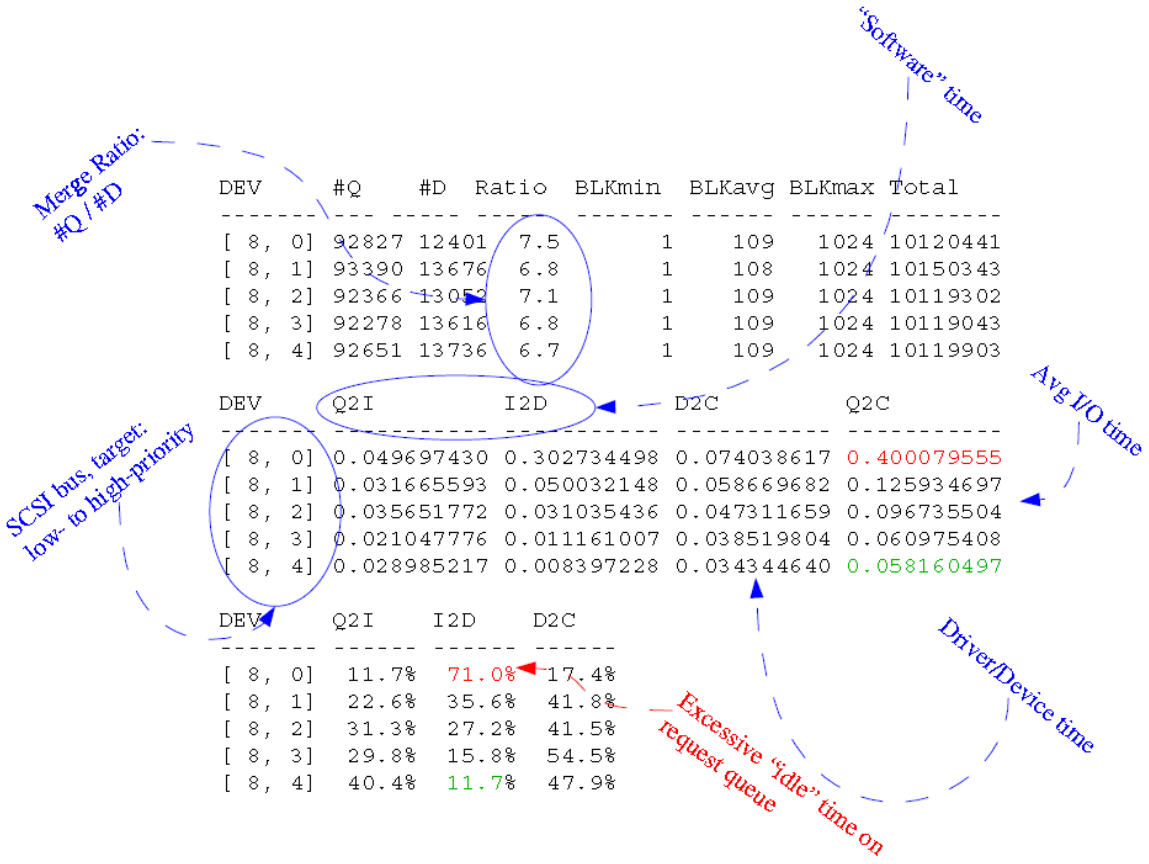
What is not visible in the application trace however is how these IO requests are being processed through the Linux IO stack down to the device driver (see Figure 1). The well-known (and well-documented) UNIX/Linux performance command *iostat* provides IO statistics (aggregated per device) at the device driver level. The iostat tool, while still very much useful in any IO performance evaluation study, does not provide any detailed performance data on a per IO request basis. Therefore, whatever happens on a per IO basis (in regards to IO wait, IO request coalescing, or IO request reordering scenarios) between the application layer and the device driver is *hidden* or *black boxed*. Fortunately, Linux 2.6 provides the *blktrace* tool set that allows scrutinizing the IO requests at the block layer and scheduler level [6]. The *blktrace* tool set basically provides detailed block layer information on an individual IO request level. The (kernel) implementation is considered low overhead (from a systems perspective), as benchmarks have shown only an approximately 2% aggregate performance degradation under rather stressful IO scenarios. Multiple logical or physical IO devices can be specified, and even filters can be used to zoom into the area of interest. To further analyze a *blktrace* output, the *blkparse* and *btt* tools can be used. Figure 3 depict the summary of a blktrace session.

Figure 3: Linux 2.6 blktrace summary



To reiterate, *blktrace* infrastructure enables the Linux community to analyze scalability issues within the block I/O layer, as well as to quantify the potential overhead (performance cost) induced by a (as an example) software RAID solution. Further, the efficiency of various IO hardware configurations can be quantified. The tool set is also being used to determine the best possible IO scheduler configuration for a certain environment/workload, as the tool set highlights the application IO behavior/cost at the Linux bio to IO scheduler interface. Further, the tool set has been used to benchmark the (application specific) IO performance in conjunction with different Linux file system solutions. As every Linux file system (such as ext4, JFS, XFS, or Btrfs) has a different communication behavior with the Linux bio layer (see Figure 1), the tool set reflects a very powerful mechanism to determine the most efficient and effective file system setup for a particular application and HW environment. Next to the *blkparse* command, the *btt* tool aids in further analyzing the IO behavior [7]. The btt output discusses (in a very detailed manner) the IO events in the Linux 2.6 bio layer (see Figure 4). Further, the btt command allows displaying the IO behavior (on a time line) in a graphical form.

Figure 4: Linux 2.6 btt summary

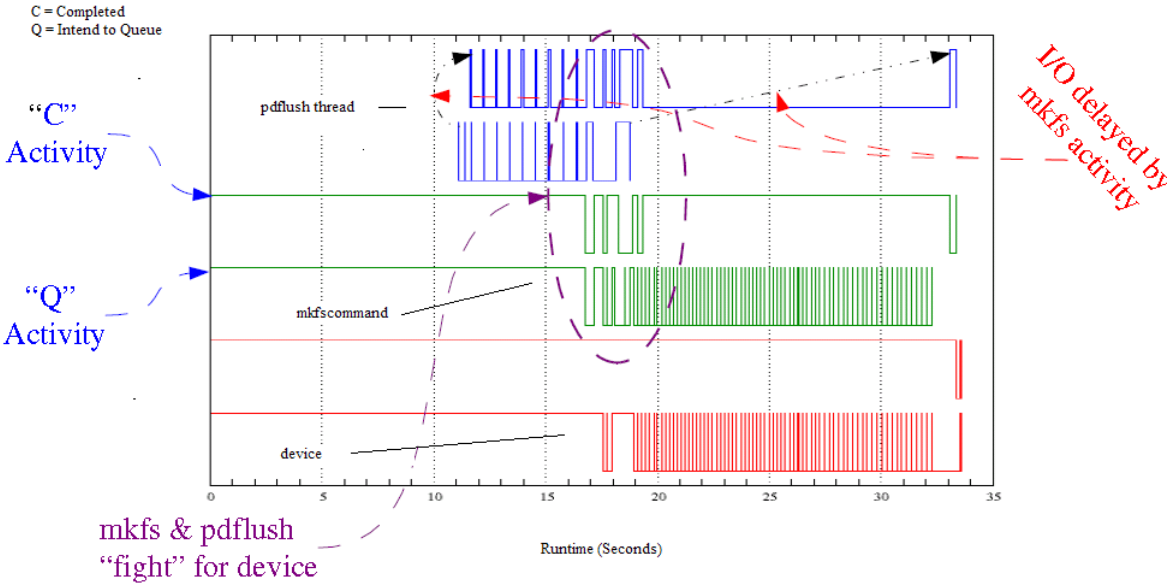


As depicted in Figure 4, *btt* generates IO data, indicating ranges where threads and devices are actively handling various IO events such as block I/O entered, I/O inserted/merged, I/O issued, or I/O completed [7]. To reiterate, the Linux 2.6 *blktrace* tool set fills the gap between the application space and the actual device driver when analyzing IO performance. The tool set is being used to optimize the application driven IO demand onto the file system, the IO scheduler, and ultimately onto the HW IO subsystem. Next to identifying bottlenecks and addressing scalability issues in the Linux 2.6 IO kernel framework, the tool set supports choosing the file system, as well as the IO scheduler that is most suited for a certain environment.

Figure 5 depicts a btt output on a time-line. In this example, the blktrace output was further refined via btt on a system that showed the following behavior. Some IO tasks disclosed a large lag time between being inserted into the bio and IO scheduler framework and actually being submitted to the device driver. During this time, the system was in the process of creating file systems on single devices. Figure 5 shows the time line for the Linux *pdflush* daemon and the *mkfs* command. The graph reveals a steady stream of IO requests for the mkfs command (at the device layer), whereas for the pdflush daemon, a time lag of

approximately 14.5 seconds exists. For the pdflush daemon, at the data point 19 seconds, the last batch of IO requests enters the Linux IO bio layer, but are not completed until the data point 33.5 (see Figure 5). The analysis of the system showed that the server was setup/configured with the anticipatory IO scheduler, allowing the mkfs command to proceed while holding back the pdflush's IO requests. Switching the IO scheduler to noop resulted in a much more balanced IO behavior between the mkfs and the pdflush IO requests.

Figure 5: Linux btt output on a time-line (mkfs vs. pdflush IO behavior)



6.0 CFQ IO Scheduler Optimization

As already discussed in Section 4.1, in the CFQ IO scheduler framework, some IO requests may experience rather long wait times in the scheduler queue. While CFQ provides timeout values for synchronous and asynchronous IO tasks, the default values of 125 milliseconds and 250 milliseconds, respectively, are in IO terms rather long. Hence, depending on the IO workload mix, some IO requests may experience *starvation* inside the CFQ IO scheduler, affecting aggregate application response time. Further, the CFQ quantum parameter (default value is 4) governs the number of dispatched IO requests to the device queue. Most of the CFQ tunables are associated with an IO priority value. In this section, an Artificial Neural Network (ANN) based algorithm is proposed that addresses the current CFQ starvation issues by optimizing the per queue quantum value. To illustrate, while a queue is selected for processing, the quantum is changed utilizing an ANN algorithm.

The ANN recognizes trend information of time series data by analyzing the IO behavior against previous states [11]. The input values into the ANN are the quantum of the queues (now dynamic), as well as the average response time values. An actual relationship between the quantum change of a specified queue with the average response time and the quantum of other queues is established. Ergo, the ANN determines the quantum of a specified queue by using the optimized quantum behavior of other queues in the system. To illustrate, the proposed algorithm determines the current optimal quantum value in relation to the average response time *by optimizing the average response time of the queue*. The learning phase of the ANN revolves around the response time values per queue. For a particular queue, if the new (calculated) average response time is less than the last calculated response time, the new value is chosen. If the new average response time is greater than the last value, the last value is kept as the optimized value, and the calculated quantum is selected for the queue. If the quantum of another queue changes (workload dependent), a (potentially) new quantum may have to be established. From an implementation perspective, a Hopfield network was chosen [10]. A Hopfield network is considered as iterative auto-associative memory. John Hopfield (a physicist) introduced a simple (in structure), but very effective artificial neural network. A Hopfield network is normally used for auto-association problems (vector based). A Hopfield network reflects a single layer network with a number of neurons equal to the number of inputs *Xi*. The network is fully connected, which implies that every neuron is connected to every other neuron, and hence all the inputs are

also the outputs (feedback based). Feedback is one of the key features of the Hopfield network, and this feedback is the basis for the convergence to the correct result. The learning algorithm for a Hopfield network is based on the Hebbian rule, and is basically a summation of products. The Hebbian Learning Rule is a learning rule that specifies how much the weight of the connection between two units should be increased or decreased in proportion to the product of their activation. However, since the Hopfield network has a number of input neurons, the weights are no longer a single array or vector, but a collection of vectors, which are most compactly contained in a single matrix [10].

To assess and quantify the performance potential of the proposed extension to the CFQ IO scheduler, a set of IO benchmarks were executed against a RAID setup consisting of 8 Seagate drives. For all the benchmarks, XFS file systems were used. The changes to the CFQ IO scheduler were incorporated into a Linux 2.6.30 kernel. For the Linux RAID benchmarks, a RAID-10 setup (with 8 disks) was configured. The hardware setup for all the conducted benchmarks can be summarized as:

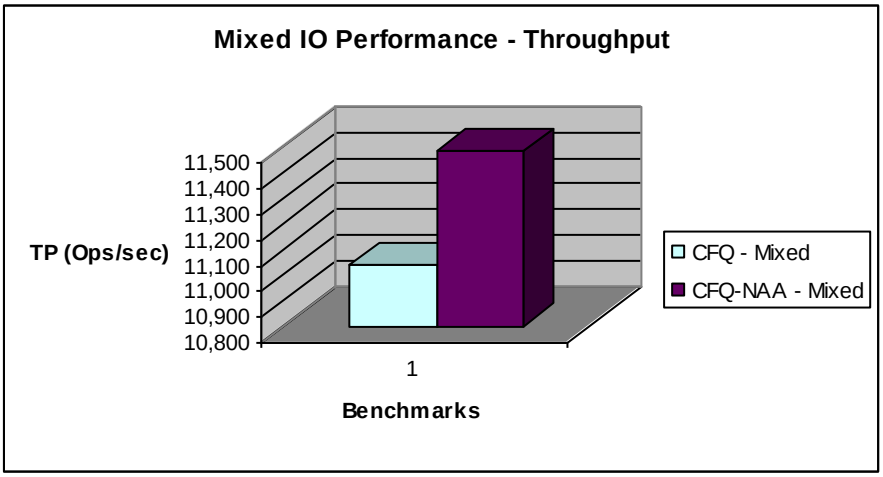
- *2 Dual-Core Xeon Processors (2.5GHz)*
- *16GB RAM (to avoid excessive caching scenarios, only 4GB were used for the benchmarks)*
- *Areca ARC-1220 SATA PCI-express 8x controller with 8 Seagate 7,200RPM 500GB SATA II drives*

All the empirical studied were executed by utilizing the Flexible File System Benchmark (FFSB) IO benchmarking set. FFSB represents a sophisticated benchmarking environment that allows analyzing performance by simulating basically any IO pattern imaginable. The benchmarks can be executed on multiple individual file systems, utilizing an adjustable number of worker threads, where each thread may either operate out of a combined or a thread-based IO profile. Aging the file systems, as well as collecting systems utilization and throughput statistics is part of the benchmarking framework. For the RAID setup (8 concurrent IO threads), the following benchmarks were conducted:

- *Sequential Read*, 40MB files, 4KB read requests
- *Sequential Write*, 100MB files, 4KB write requests
- *Random Read*, 40MB files, 4KB read requests, 1MB random read per IO thread
- *Random Write*, 40MB files, 4KB write requests, 1 MB random write per IO thread
- *Mixed*, 60% read, 30% write, 10% delete, file sizes 4KB to 1MB, 4KB requests

Each benchmark was executed 15 times, and the results reflect the mean values over the sample set. For all the conducted benchmarks, the coefficient of variation (CV) was less than 4%, hence the statement can be made that all the runs are reproducible with the resources at hand. For all the benchmarks, an efficiency value, representing the number of IO operations divided by the CPU utilization was calculated. In other words, the efficiency value fuses the throughput potential and the corresponding CPU demand/consumption. To simulate a heterogeneous IO environment, the priority of the IO threads was chosen based on a pseudo random number generator.

Figure 6: Mixed Worklaod – Throughput



The results of the conducted FFSB benchmarks disclosed a very encouraging throughput behavior. For the sequential IO benchmarks, the ANN based CFQ IO scheduler improved the throughput by approximately 3%. For the random IO benchmarks, the throughput was improved by approximately 1%. The largest throughput improvement was reported for the mixed workload pattern with approximately 4% (see Figure 6). For none of the benchmarks conducted for this study, any performance degradation with the ANN CFQ scheduler was reported. Having said that, for all the conducted benchmarks, the CPU utilization with the ANN based CFQ scheduler increased and hence, the CFQ ANN results report a lower efficiency value. For the sequential IO benchmarks, the efficiency value for the ANN CFQ setup is approximately 8.5% lower, whereas for the random IO benchmarks, the efficiency value for the ANN CFQ setup dropped by approximately 7%. For the mixed workload benchmark, the ANN CFQ value was lower than the current CFQ setup by approximately 7.5% (see Figure 7).

Figure 7: Mixed Workload – Efficiency Value

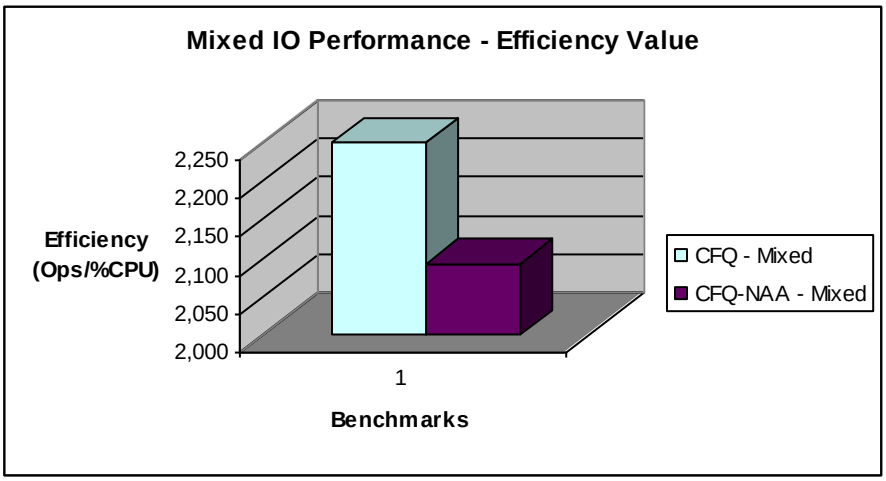


Table 1 summarizes the results of the conducted benchmarks. The increase in CPU utilization is due to the ANN processing tasks to determine the optimal quantum per queue in relation to the average response time. A more CPU efficient implementation of the Hopfield network (code optimization) should result in better CPU utilization values. Actual *blktrace* data taken throughout the benchmark study revealed for the CFQ ANN scheduler an improved (more balanced) IO processing behavior, resulting into a more optimized average response time per queue. Ergo, at least for the benchmarks conducted for this study, the proposed ANN algorithm addresses the IO starvation issues discussed in this paper.

Table 1: Summary of the conducted FFSB benchmarks

Sequential Writes (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
CFQ	216.6	1,600
CFQ-ANN	223.098	1,472
Sequential Reads (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
CFQ	257.1	15,500
CFQ-ANN	264.813	14,105
Random Reads (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
CFQ	9.1	2,580
CFQ-ANN	9.191	2,399
Random Writes (8 threads)	*TP MB/sec*	*Efficiency (Ops/%CPU)*
CFQ	40.1	2,100
CFQ-ANN	40.501	1,953
Mixed Worklaod (8 threads)	*Ops/second*	*Efficiency (Ops/%CPU)*
CFQ	11,040	2,250
CFQ-ANN	11,482	2092.5

7.0 Conclusions & Acknowledgments

The main perspective of this paper was on introducing the (rather complex and unique) Linux 2.6 IO stack, focusing on the block IO layer and the available IO schedulers. A layered methodology that is based on Linux 2.6 trace tools to quantify, analyze, and ultimately optimize Linux 2.6 IO performance was elaborated on. The discussed (potential) IO request starvation issues that are possible in a CFQ IO scheduler environment were targeted via the introduction of a Hopfield network based algorithm that addresses the (static) quantum behavior in the IO scheduler. For the benchmarks conducted for this study, the results disclosed a slightly higher throughput potential and a more balanced IO behavior under increased CPU utilization. While the proposed ANN algorithm showed potential, code optimization and a much more comprehensive benchmark study has to be conducted to reach any conclusion on the usefulness of the suggested approach.

Jens Axboe (Oracle) maintains the Linux 2.6 block IO layer. Next to all his contributions to the CFQ scheduler, he also developed the blktrace tool. Allan Brunelle (HP) significantly contributed to the rapid evolvement and acceptance of the blktrace tool set by the Linux community. Some of the graphs shown in Section 5 are courtesy of the HP scalability and performance group (Allan Brunelle). Next to these 2 Linux OS engineers, there are too many others to name here, engineers who contributed countless hours and hundreds of lines of code to optimize the Linux 2.6 IO framework.

References

1. Axboe, J., “Deadline I/O Scheduler Tunables”, SuSE, EDF R&D, 2003
2. Axboe, J., “Linux Block I/O Layer – Present & Future”, OLS 04, Ottawa, 2004
3. McKenney, P., “Stochastic Fairness Queueing”, INFOCOM, 1990
4. Mosberger, D., Eranian, S., “IA-64 Linux Kernel, Design and Implementation”, Prentice Hall, NJ, 2002
5. Piggin, N., “Anticipatory I/O Scheduler”, UNIX Source Code Documentation (as-iosched.txt), 2004
6. Brunelle, A. “Gelato”, HP Scalability & Performance Group, 2006
7. Brunelle, A. “The btt User Guide, HP Scalability & Performance Group, 2007
8. Johnson, S. “Performance Tuning for Linux Servers”, IBM Press, 2005
9. Heger, D. “Quantifying IT Stability – 2nd Edition”, ISBN 978-0-578-05264-9, 2010
10. Gamil, A. “Mapping the Three Matching Problem into Hopfield Neural Networks”, Qassim University, 2008
11. Gavrillov, A., Kangler, V. “The Use Of Artificial Neural Networks For Date Analysis”, Novosibirsk University, 2001

Appendix A; Linux 3.x IO Subsystem

The next few paragraphs discuss the 3 IO schedulers available in the Linux 3.x IO framework (it has to be pointed out that the anticipatory IO scheduler has been dropped as of Linux kernel version 2.6.33). Each IO scheduler has a different performance behavior, and hence, the internal working of each scheduler has to be known when designing and implementing Linux server systems.

CFQ IO Scheduler (the Linux 2.6/3.x Default IO Scheduler)

The CFQ scheduler operates by placing synchronous requests that are submitted by threads into a number of per-thread queues, and then by allocating actual time-slices for each of the queues to access the (physical) IO subsystem. The length of the time-slice, as well as the number of IO requests that a queue is allowed to submit depends on the IO priority of the given thread. Asynchronous IO requests for all threads are bundled together (priority based) in a few queues. By design, good aggregate throughput behavior is achieved by allowing a thread queue to idle at the end of a synchronous IO batch, thereby allowing the IO framework to *anticipate* (short-term) close IO requests from the same thread (similar to the deadline IO scheduler discussed below). This behavior is considered a natural extension of granting actual IO time slices to a thread. The designers behind the CFQ scheduler created the concept of having actual IO queues for each thread. These queues are created ad-hoc for each particular thread. Further, the designers segregated the concept of IO into two distinct sections, synchronous and asynchronous IO. Synchronous IO is important as the application threads have to stall until the IO request completes. To illustrate, if an application *read()* request does not hit in either the cache or the memory subsystem, the application *read()* request has to block until the data is available in memory, a scenario that obviously has a profound impact on application performance. In most circumstances, *write()* requests are asynchronous. Ergo, the *write()* requests are pushed into the memory subsystem, if possible consolidated, and flushed to disk either time-based, or based on the state of the memory subsystem.

Next to distinguishing between synchronous (which are favored) and asynchronous IO operations, the CFQ design also *prefers* read over write operations. As already discussed, read requests have the potential to stall application processing, impacting aggregate systems performance. Further, read requests, based on the elevator approach, may starve other read requests that disclose (from an IO geometry perspective) long-distance cases (based on the currently processed IO batch). Therefore, favoring read over write operations (in some circumstances) may improve aggregate read IO responsiveness, and reduce the probability of read IO starvation. Based on the dynamic nature of IO operations (and the corresponding priority scenarios) though, there is always a possibility that an IO request gets pushed back in a queue, and hence gets delayed. To combat such a behavior, in CFQ, each thread is associated with a time-out value. If the IO thread is not executed within that epoch, as the time expires, the IO thread is instantly scheduled for execution. As with all Linux 2.6 IO schedulers, the designers of the CFQ IO scheduler exported several tuning parameters into user space. In other words, the actual performance behavior of the CFQ scheduler, based on the application workload and the setup of the physical IO subsystem, can be adjusted/modified by the user community.

The noop IO Scheduler

The noop scheduler inserts the set of I/O requests into a simple, unordered FIFO queue and only provides simple IO request merging. The noop scheduler is very effective in environments where IO performance optimization is incorporated in a lower layer of the (physical) IO stack. In other words, the noop scheduler is normally chosen in cluster environments that have SAN access. Further, the noop scheduler is best used with solid state disks (SSD) or any other device that can not benefit from re-ordering of multiple I/O requests at the Linux IO scheduler level.

The Deadline IO Scheduler

The deadline I/O scheduler incorporates a per-request expiration-based approach and operates on 5 I/O queues. The basic idea behind the implementation is to aggressively reorder requests to improve I/O performance while simultaneously ensuring that no I/O request is being starved. More specifically, the scheduler introduces the notion of a per-request deadline, which is used to assign a higher preference to *read()* than *write()* requests. The scheduler maintains the 5 I/O queues. During the *enqueue* phase (see Figure 1A), each I/O request gets associated with a deadline, and is being inserted in I/O queues that are

either organized by the starting logical block number (a sorted list) or by the deadline factor (a FIFO list). The scheduler incorporates separate sort and FIFO lists for *read()* and *write()* requests, respectively. The 5th I/O queue contains the requests that are to be handed off to the device driver. During a *dequeue* operation, in the case where the dispatch queue is empty, requests are moved from one of the 4 (sort or FIFO) I/O lists in batches. The next step consists of passing the head request on the dispatch queue to the device driver (this scenario also holds true in the case that the dispatch-queue is not empty).

The logic behind moving the I/O requests from either the sort or the FIFO lists is based on the scheduler's goal to ensure that each *read()* request is processed by its effective deadline, without starving the queued-up *write()* requests. In this design, the goal of economizing the disk seek time is accomplished by moving a larger batch of requests from the sort list (logical block number sorted), and balancing it with a controlled number of requests from the FIFO list. Hence, the ramification is that the deadline I/O scheduler effectively emphasizes average *read()* request response time over disk utilization and total average I/O request response time. Benchmarks have shown that some database applications do perform very well with the deadline IO scheduler.

Figure 1A: Linux 2.6/3.x I/O Stack