Efficient and Adaptive Proportional Share I/O Scheduling

Abstract

In most data centers, terabytes of storage are commonly shared among tens to hundreds of hosts for universal access and economies of scaling. Despite much prior research, most storage arrays do not provide useful mechanisms to provide Quality of service (QoS) guarantees to applications. Storage administrators are often forced to isolate workloads using expensive storage provisioing. We believe that lack of adoption of QoS mechanisms is mainly due to the low I/O efficiencies that result from the various proposed QoS techniques. This motivates the need for more flexible QoS mechanisms since the efficiency of I/O devices depends crucially on the order in which the requests are served. In this paper, we attempt to alleviate the I/O efficiency concerns of proportional share schedulers. We first provide a framework to study the inherent trade-off between fairness and I/O efficiency. We propose two main parameters - *degree of concurrency*and *batch size* as control knobs. We find that significantly *higher I/O efficiency* can be achieved by slightly *relaxing short-term fairness* guarantees. We then present a self-tuning algorithm that achieves good efficiency while still providing fairness guarantees. The algorithm doesn't require any workload-specific parameters to operate. Experimental results indicate that an I/O efficiency of over 90% is achievable by allowing the scheduler to deviate from proportional service for a few seconds at a time.

1 Introduction

Increasing cost pressures on IT environments have been fueling a recent trend towards storage consolidation, where multiple applications share storage systems to improve utilization, cost, and operational efficiency. The primary motivation behind storage QoS research has been to alleviate problems that arise due to sharing, such as handling diverse application I/O requirements and changing workload demands and characteristics. For example, the performance of interactive or time-critical workloads such as media serving and transaction processing should not be hurt by I/O intensive workloads or background jobs such as online analytics, file serving or virus scanning. Despite much prior research [3, 16, 17, 19], QoS mechanisms do not enjoy widespread deployment in today's storage systems. We believe this is primarily due to the low I/O efficiencies of the existing QoS mechanisms.

Fairness and I/O efficiency are known to be quite difficult to optimize simultaneously for multiple applications sharing I/O resources. Disk schedulers usually attempt to maximize overall throughput by reducing mechanical delays while serving I/O requests. The individual application (or process) making the I/O request is usually not considered in the scheduling decisions. The conventional belief is that maximizing the overall throughput is good for all applications accessing the storage system, and providing fairness can substantially reduce the overall throughput. In contrast, other resources in a system such as CPU, memory, and network bandwidth can be multiplexed based on per-process behavior by using well known fairness algorithms [2, 5, 7, 18, 20]. We believe that the existing QoS mechanisms proposed so far have not adequately addressed the problem of losing I/O efficiency due to proportional sharing. The main issue at hand is to find the right balance between the two opposing forces of proportional share guarantees and IO efficiency. A QoS mechanism for storage systems should have the following properties:

- **Fairness guarantees:** to provide proportional share fairness to different applications.
- **I/O efficiency:** to achieve high I/O efficiency comparable to workloads running in isolation.
- **Control knobs:** to provide the ability to control the inherent trade-off between the I/O efficiency and the proportional share guarantees.
- **Work conservation:** storage system is not kept idle when there are pending requests.

In this paper, we first provide a framework to systematically study the trade-off between fairness and efficiency and propose simple mechanisms to adaptively tune the system to find a balanced operation point. We find that I/O efficiency can be improved if we relax the *fairness granularity*, the minimum time interval over which the QoS mechanisms guarantee fairness in proportional shares of the contending applications. This is significant as it indicates that it may be possible to improve the I/O efficiency without greatly affecting the QoS guarantees.

While the tension between fairness and efficiency is well known, our second main contribution lies in providing a novel online, adaptive mechanism to improve the I/O efficiency of proportional share schedulers. We propose two control mechanisms to achieve these properties: *variable size batching* and *bounded concurrency*.

A batch is a set of requests from an application that are issued consecutively without intervention from other applications. Concurrency refers to the number of requests outstanding at the device at any given time. The first mechanism allows each application to choose a batch size appropriate to its access characteristics. This is useful for workloads that exhibit spatial locality as it reduces the delays due to excessive disk seeks. The second mechanism allows the fair-share scheduler to keep a sufficient number of pending requests (*i,e,* level of concurrency) at the device, so that existing seek-optimizing schedulers can provide higher throughput. However, the concurrency needs to be bounded if the fairness guarantees are to be honored. We show that these two mechanisms are effective in improving I/O efficiency while only slightly increasing the fairness granularity for the QoS guarantees. We also develop an algorithm that adapts the settings of these two parameters based on the workload characteristics. This is useful as it allows us to keep the I/O efficiency high in the presence of dynamically changing workloads without impacting the QoS guarantees (*i.e.,* the fairness granularity). Our analysis shows that in worst case, the fairness granularity is a linear function of these two parameters. Previous work by Jin *et al.*(SFQ(D) [12]) has only addressed the impact of the concurrency parameter on fairness.

In the remainder of this paper, we first discuss the prior work in section 2 and describe our system model in section 3. Then, we describe our mechanisms to trade off between I/O efficiency and the fairness granularity in section 4, and develop analytical bounds for the fairness granularity in section 5. We evaluate our approach in section 6 and then conclude.

2 Related Work

Providing QoS support has been an active area of research in systems and many proposed mechanisms in the networking domain have found their way into deployments. For example, (WFQ [5], WF^2Q [2], SFQ [6,8,9], DRR [18]) have been adopted for traffic shaping and providing fairness for network link bandwidth. Some variants of these algorithms have also been proposed for storage systems. Existing approaches for *QoS in storage* can be classified into three main categories: (1) Fair queuing based scheduling algorithms, (2) time slicing at the device, and (3) control theoretic approaches.

Fair queuing based techniques use variants of the WFQ algorithm [5] to provide QoS. YFQ [3], SFQ(D) [12], Avatar [23], and Cello [17] use virtual time based tagging to select IOs and then use a seek optimizer to schedule the chosen requests. Some of these techniques have proposed using high concurrency at the storage server for higher throughput. However, none of them have studied the impact of such optimizations on fairness. Stonehenge [11] and SCAN-EDF [16] also consider both seek times and request deadlines. Other approaches such as *p*Clock [10] do burst handling and provide fair scheduling to handle both latency deadlines and bandwidth allocation. A fundamental limitation of existing techniques is that they focus mainly on fairness and do not evaluate the trade-off between fairness and I/O efficiency. Our work extends one such algorithm to support a balance between fairness and efficiency.

Among the scheduling-based techniques, Zygaria [21] and AQuA [22] use hierarchical token buckets to support QoS guarantees for distributed storage systems. Zygaria supports throughput reserves and throughput caps while preserving I/O efficiency, but it neither provides mechanisms for trading fairness with efficiency nor adapts scheduling based on the workload. Similarly, the ODIS scheduler in AQuA employs a "bandwidth maximizer" that attempts to increase aggregate throughput as long as the QoS assurances are not violated. While ODIS employs a throttling-based heuristic algorithm that adjusts the token rate based on overall disk utilization, it does not consider individual workload characteristics. In cases where the system is over-loaded and not all QoS requirements can be met, there is no guarantee of proportional service. No special effort is made to maintain the efficiency of sequential and spatially local workloads. By contrast, our framework guarantees that, when workloads are backlogged, the service will be allocated proportionately between the workloads based on their weights; this guarantee is proven theoretically and demonstrated experimentally. In addition, our mechanism enables high I/O efficiency for spatially local workloads by trading off fairness granularity - i.e., by allowing brief deviations from proportional service.

Other techniques such as Fahrrad [15] and Argon [19] are based on IO time multiplexing at the disk. This has the advantage of preserving the IO access patterns of an application and avoiding interference with other workloads. Fahrrad [15] tries to provide real time guarantees for disk time utilization. Reserving disk time allows one to charge based on IO workload, which can be quite useful in some cases. In most storage systems the concurrency of multiple IOs pending at the same time, makes it very difficult to estimate per IO completion time. In Argon [19], each application is assigned a time quantum dedicated to its IO requests. One issue with this approach is the potential for increased latency. IO requests from an application that miss the application's timeslice (either because they did not complete during the timeslice, or arrived after it ended) must wait until the next timeslice to receive service. The worst case latency bounds increase with the number of applications and the duration of the time quantum. Secondly, during a timeslice the server sees only the requests from the

single application scheduled in that interval. While this improves the efficiency of serving sequential requests, it decreases the effectiveness of the seek optimizer for random requests, because it does not take all the pending requests into consideration.

Control theoretic approaches such as Triage [13] and Sleds [4] use client throttling as a mechanism to ensure fair sharing among clients and may lead to lower utilization. Façade [14] tries to provide latency guarantees to applications by controlling the length of disk queues. This can lead to lower overall efficiency and the tradeoff between the loss of efficiency and latency is not explored.

3 System Model

Our system consists of a storage server that is shared between a number of applications. Each application has an associated *weight*. The goal of the proportional share (fair) scheduler is to provide active applications I/O throughput in proportion to their associated weights, while maintaining high efficiency. The *fair scheduler* is logically interposed between the applications and the storage server. In an actual implementation, it could reside in the storage server, in a network switch, in a separate "shim" appliance [12], or in a device driver stack. The fair scheduler maintains a set of input queues, one for each application, and an output queue. Requests are scheduled from the input queues to the output queue and then they are scheduled on to underlying devices based on some seek optimizing criterion. Similar two level architectures have also been proposed in earlier works (see Section 2, where the first level does fair scheduling and the second level does seek optimization.

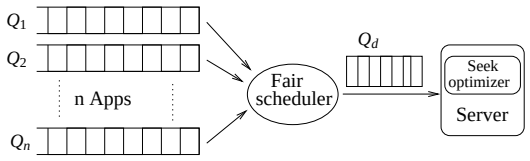


Figure 1: System Model

Notation: The number of applications is denoted as N. The i^{th} application is a_i; its weight is w_i, and its queue in the fair scheduler is Q_i. D is the number of outstanding scheduled requests - *i.e.*, the number of requests in the scheduler output queue plus those outstanding at the storage server. These and other notations we use are summarized in Table 1 for convenient reference.

3.1 Metric Definitions

The objective of our system is to provide throughput to applications in proportion to their weights, while maintaining high overall system throughput. The performance of a storage server depends critically upon the order in which the requests are served. For example, it is substantially more efficient to serve sequential I/Os together. This is unlike other domains, such as networking, where the order in which packets are dispatched does not affect the overall throughput of a switch. For this reason, it is important to measure the overall throughput (efficiency), in addition to a fairness criterion. Efficiency denotes the ratio of the actual system throughput to that attained when the applications are run without interference. Fairness refers to how well the application throughputs match their assigned weights.

SYMBOLS	DESCRIPTION
N	number of applications
a_i	the *ith* application
w_i	weight of application a_i
Q_i	fair scheduler queue for a_i
G_i	batch size for application a_i
D	number of outstanding scheduled requests
$n_i(t_1,t_2)$	throughput for application a_i, alone
$r_i(t_1,t_2)$	throughput for application a_i, shared
$\mathscr{E}(t_1,t_2)$	efficiency of the scheduler
$\mathscr{F}(t_1,t_2)$	fairness of the scheduler

Table 1: **Notation used in this paper.** The last four metrics are defined over a time interval (t_1,t_2). For notational convenience we omit (t_1,t_2), since the time interval is implicit.

We first define an efficiency measure that captures the slowdown due to scheduling the mix of requests rather than running them in isolation. To motivate the definition, consider two applications a_1 and a_2 which have isolated throughputs of $n_1 = 100$ and $n_2 = 200$ (requests/sec) respectively. Suppose that when run together using a fair scheduler, 25 requests of a_1 and 40 requests of a_2 were completed in an interval of $T_s = 1$ second. Now, if these requests of a_1 were run in isolation (at a rate of 100 req/sec) they would complete in 0.25 sec; similarly the 40 requests of a_2 would complete in 0.2 sec. Hence the total time to complete requests of both applications using an isolating scheduler would be $T_m = 0.45$ sec. The efficiency of the fair scheduler is $T_m/T_s = 0.45$. If the fair scheduler were improved and the measured throughputs of a_1 and a_2 increased to 40 and 80 req/sec, the efficiency would increase to $(40/100+80/200)/1 = 0.8$. In some cases the use of a fair scheduler can actually lead to a speedup rather than a slowdown by merging the workloads; in this case the efficiency can exceed 1. For instance, if the measured throughputs were 60 and 120 req/sec, the corresponding efficiency would be $60/100+120/200 = 1.2$.

Definition 1 provides a formal definition for the efficiency measure discussed above. Lemma 1 derives a

simple relation between efficiency and the measured and isolated throughputs of the applications.

Definition 1. *Efficiency metric ($\mathscr{E}$): Let S be a set of requests serviced together in the interval (t_1,t_2) by the fair scheduler. Let $T_s = (t_2 - t_1)$. Let T_m denote the total time needed to service the requests in S by running each application in isolation. The efficiency of the scheduler in the interval (t_1,t_2) is defined as:*

$$\mathscr{E}(t_1,t_2) = T_m/T_s \tag{1}$$

Lemma 1. $\mathscr{E}(t_1,t_2) = \sum_i r_i/n_i$

Proof. Consider the time interval (t_1,t_2) and suppose the fair scheduler services β_i requests of a_i, $i = 1,\cdots n$, when running all the applications together. $T_s = t_2 - t_1$ denotes the length of the interval. By definition, the throughput of a_i in the shared environment r_i=β_i/T_s. The time required to service the β_i requests of a_i in isolation is given by $t_i' = \beta_i/n_i$, since n_i is the throughput of a_i when running in isolation. The time needed to service all the requests in S by running each application in isolation is therefore given by $T_m = \sum_i t_i' = \sum_i \beta_i/n_i = T_s \sum_i r_i/n_i$. Hence efficiency $\mathscr{E}(t_1,t_2) = T_m/T_s = \sum_i r_i/n_i$. □

Higher values are better for this metric and and a value of 1 means that the throughput obtained for a given workload matches that obtained by running the different applications making up the workload in isolation. A value greater than 1 means that the concurrent workload has higher throughput than running the applications in isolation. This happens when random workloads are merged as shown in the experimental results in Section 4.1. This is because the lower level seek optimizer gets more opportunities to reduce the time spent on seeking. Also note that our definition is independent of the weight of the applications. In previous works such as Argon [19], the notion of efficiency is coupled with the notion of fairness, where the expected output for an application with weight w_i is $w_i \times n_i$. Here w_i is the normalized weight, such that all of the weights sum to 1. The main benefit of our metric is that it allows us to compare efficiencies even for schedulers with widely different fairness guarantees.

We next define a fairness index that measures how close the ratios of the throughputs of the different applications comprising the workload matches the ratios that would result from a proportional allocation. Over the interval (t_1,t_2), let the fair scheduler provide a throughput of r_i for application a_i. Then $w_i' = r_i/\sum_j r_j$ is the fraction of the throughput that a_i receives in the shared server. By definition, the weight w_i is the fraction of the throughput that a_i should receive from an ideal fair scheduler. $W' = [w_1', w_2', \cdots w_N']$ denotes the vector of *measured* service fractions and $W = [w_1, w_2, \cdots w_N]$ the vector of *ideal* service fractions expected from a fair schedule. The measure of fairness is the "distance" between the measured vector W' and the ideal vector W. A number of different distance measures are discussed in the statistics literature; we use the well-known L_1 norm (Manhattan distance) as the measure in this paper. The L_1 distance between the vectors is defined as $\sum_i |w_i - w_i'|$. Note that since $\sum_i w_i = 1 = \sum_i w_i'$, both W and W' are unit vectors under the L_1 norm.

Definition 2. *Fairness index ($\mathscr{F}$): Let application a_i obtain a throughput r_i over an interval (t_1,t_2). The total throughput is $R = \sum_{i=1}^{n} r_i$, and the measured weight of a_i is $w_i' = r_i/R$. The fairness index is defined as:*

$$\mathscr{F}(t_1,t_2) = \sum_i |w_i - w_i'| \tag{2}$$

Note that the L_1 distance between the vectors, and hence $\mathscr{F}(t_1,t_2)$, can range between 0 and 2. The lower value is better, since it means that the ratio of the application throughputs have a good match with the weights.

Finally, we consider the notion of fairness granularity. A scheduler that is fair over short intervals of time is also fair over large intervals (by simple aggregation), but the reverse is not necessarily true. As such, a scheduler that is fair over short intervals is strictly fairer than one that is only fair over long intervals. Intuitively, the fairness granularity of a scheduler is the smallest length of time over which it is consistently fair; smaller is better. Thus, a scheduler with a fairness granularity of one second may deviate from a proportional allocation of service over intervals shorter than one second, but assures proportional allocation for measurement intervals of one second or longer. The techniques we propose in the next section work by relaxing fairness granularity in order to gain efficiency. A formal definition of fairness granularity is given below.

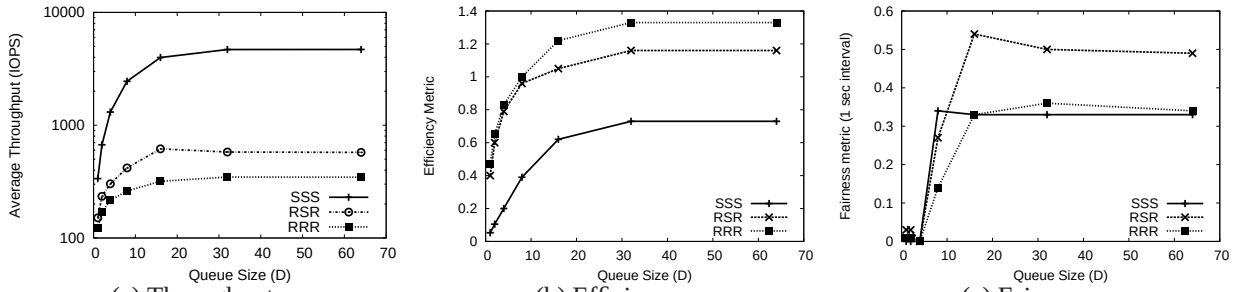
Definition 3. *Fairness Granularity $\delta(f_m)$ is defined as the smallest time duration ε such that 95^{th} percentile value of the set $\{\mathscr{F}(t_1 + (m-1)\varepsilon,\ t_1 + m\varepsilon),\ m = 1,\cdots (t_2 - t_1)/\varepsilon\}$ is less than f_m.*

That is, the Fairness Granularity is the smallest interval length ε, for which at least 95% of the intervals have a fairness index less than f_m.

Having looked at the metrics that we use to measure the performance of a fair scheduling framework, we now look at various fair scheduling algorithms and the design of an efficient fair scheduler.

4 Fair Scheduler Design

In this section, we first study the inherent trade-off between the I/O efficiency and the fairness guarantees of proportional share I/O schedulers and introduce two parameters that impact both. We characterize this trade-off



(a) Throughput (b) Efficiency (c) Fairness

Figure 2: *Bounded concurrency.*

experimentally by modifying the I/O issue behavior of a proportional share scheduler and using synthetic workloads. We then incorporate our findings into a new design for an efficient proportional share I/O scheduler.

For our experimental evaluation, we used a modified version of the Deficit Round Robin (DRR [18]) scheduler. The basic DRR algorithm performs scheduling decisions in rounds: it allocates a *quantum* of tokens to each application (or input queue) in a round, and the number of tokens is proportional to the application's weight. The number of IOs transferred from an application's input queue to the output queue is proportional to the number of accumulated tokens the application has. If the application has no IOs pending in its input queue in a round, the tokens disappear. Otherwise, if there are both IOs and tokens left, but there are not enough tokens to send any more IOs, then the tokens persist to the next round (this is the deficit). The DRR algorithm can produce throughput proportional to the application's assigned weight, where the throughput is measured either in bytes/sec, or in IOs/sec (IOPS), by changing how tokens are charged for the IOs. We use IOPS in this paper.

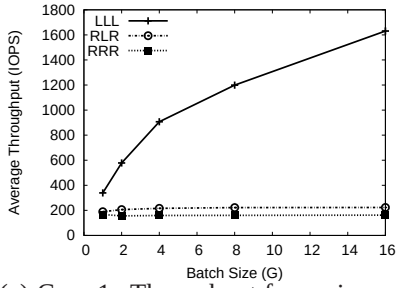
Although our adaptations can be combined with most fair schedulers such as SFQ, WFQ, *etc.*, we chose DRR for two main reasons: (1) the run-time for DRR is O(1) amortized over a number of requests, whereas other schedulers take O($\log N$) for N applications; (2) DRR provides similar fairness guarantees as other proportional share algorithms; We performed two modifications to the basic DRR algorithm so that we can study the relationship between I/O efficiency and the fairness granularity exhibited by the DRR. The first modification allows us to control the concurrency of the I/O requests at the storage system and the second one allows us to take advantage of the spatial locality of a request stream, if any. In the next two sections, we describe each of these modifications in detail and present our experimental results showing how they impact the I/O efficiency and the fairness granularity.

4.1 Bounded Concurrency

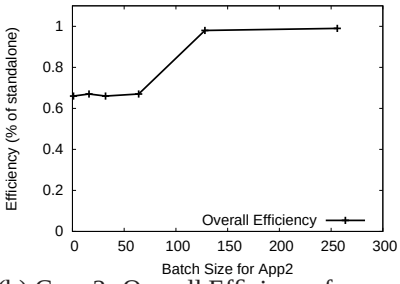
The amount of concurrency at the storage device has a profound impact on the achievable throughput. This is because higher levels of concurrency allow the scheduler to improve the request ordering so that the mechanical delays are minimized. In addition, higher levels of concurrency allow RAID devices or striped volumes to take advantage of the multiple disk drives they contain.

Proportional share I/O schedulers carefully regulate the requests from each application before issuing them to the storage system. This is necessary for achieving the desired proportionality guarantees that these schedulers seek to provide. Unfortunately, this also has the side effect of limiting the amount of request concurrency available at the storage devices. As a result, even if there is concurrency available at the workload, the DRR algorithm dispatches only a portion of the pending requests in a round, and the concurrency levels in storage systems tend to be low.

Our first modification to the DRR scheduler is to make the number of outstanding scheduled requests, D, a controllable parameter. We call this parameter *the concurrency bound*. This allows the modified DRR scheduler to keep a larger number of requests pending at the storage system. Figure 2(a) shows the I/O throughput obtained by the modified DRR scheduler as a function of the concurrency bound. For this experiment, we used three workloads and set their weights in the ratio 1:2:3. All three were closed workloads, each keeping a total of 8 requests outstanding. In the legend, S means a sequential workload and R means a random workload. Hence RRR means three random workloads running simultaneously. Figure 2(a) shows that overall throughput increases with higher concurrency levels, and the gains in I/O throughput are substantial. We also plot the efficiency metric for various values of D, as shown in figure 2(b). Note that efficiency is higher than 1 for mixes with random workloads. This is because putting together random workloads results in higher seek efficiency. On the other hand, sequential workload mix has a lower ef-



(a) Case 1: Throughput for various workload mixes with different batch sizes



(b) Case 2: Overall Efficiency for a mix of one random and one sequential workload

Figure 3: *Variable size batching.*

ficiency even at large queue depths because of frequent switching among various workloads and higher seek delays that do not occur when the sequential workloads are run in isolation.

While increasing concurrency improves the I/O efficiency, it also impacts the fairness guarantees of the proportional share I/O scheduler. Figure 2(c) shows the proportional share fairness index at a 1 second granularity for the same experiment. It shows that higher concurrency also leads to substantial loss of fairness, resulting in each application receiving substantially different throughputs from their assigned weights. We notice that the fairness starts decreasing at $D = 8$, and becomes similar to the fairness of a standard throughput maximizing scheduler as the concurrency bound approaches to $D = 20$. The modified DRR behaves like a pass through scheduler at this point and loses all its ability to regulate the throughput proportions of individual applications. **Observation:** *Random workloads benefit more from the higher concurrency in comparison to sequential workloads.*

4.2 Variable Size Batching

The other factor that impacts the I/O efficiency is the handling of spatial locality. Most storage systems implement some form of prefetching for sequential workloads which trades off additional transfer time with potential savings from fewer mechanical seeks. An I/O efficient proportional share scheduler also needs to handle sequential workloads differently to take advantage of the locality.

Our second modification to the DRR scheduler is to introduce variable size batching so that highly sequential workloads and large prefetches can be supported for efficient proportional sharing. We introduce a *batch size* parameter G, which refers to the number of IOs that are scheduled from an application in one round of DRR. This parameter can be different for each workload depending on the degree of spatial locality present; we denote the batch size for application a_i as G_i. Variable size batching allows more requests from a given application to be issued as a batch to the storage system before switching to the next application. Thus, it reduces interference among applications to benefit sequential workloads and workloads exhibiting spatial locality.

One way to increase the batch size is to increase the batch size of all applications in a proportionate manner for every round. This, however, leads to an increase in batching even for applications that may not necessarily benefit from it. To verify this we ran 3 different workload mixes, RRR, RLR, and LLL. Here L means a workload with high locality. Figure 3(a) shows the overall I/O throughput achieved from the modified DRR scheduler as the batch size is varied. **Observation:** *Workloads with high locality benefit substantially from the variable batch sizes and random workloads are almost unaffected by the batch size parameter.*

Since all workloads do not benefit from a higher batch size, we would like to be able to have different batch sizes based on the locality of the workload. We modified DRR to assign each application a number of tokens based on its batch size. Clearly, this can conflict with the assigned weight of the application; to balance this, applications with modified number of tokens should not receive any tokens for a number of rounds so as to preserve the overall proportions. We do this by skipping one or more rounds for these applications. The number of rounds to be skipped can be easily computed. For example, consider 3 applications with weights in ratio 1:2:3. Let the batch sizes be 128, 64 and 16 for applications 1, 2 and 3 respectively. Now, based on the weights and batch sizes, application 1 will get a quantum of 128 every 24 rounds, application 2 will get a quantum of 64 every 6 rounds and application 3 will get a quantum of 16 every round. Fractional allocations were not needed in this example, but they can also be handled in a similar manner.

To test that variable batch size indeed helps in improving efficiency, we experimented with 2 workloads,

one random and other sequential. Here, we varied the batch size of the sequential workload from 1 to 256. Figure 3(b) shows the overall I/O efficiency with the variable batch sizes. We observe that for small batch sizes the performance is lower (64 % of stand-alone throughput). However, for a batch size of 128, we get the desired efficiency (close to 100% of stand-alone throughput) and the overall throughput of the workloads is 1155 and 80 IOPS which is very close to half the stand alone performance (2380 and 160 IOPS).

However, the efficiency increase doesn't come for free — it adversely affect the fairness guarantees of the DRR algorithm. In effect, the assigned weights can be enforced by the modified DRR scheduler at a larger time granularity. When the batch of I/Os are issued from a workload a_i, it gets ahead of others in terms of allocated proportion of the shared system. As the DRR scheduler skips the workload a_i in the subsequent rounds, the assigned weights are reached but over a longer time interval.

4.3 Parameter Adaptation

We have discussed two techniques for balancing the efficiency and fairness provided by a storage server: variable size batching and bounded concurrency. Variable size batching requires a batch size per application that depends on how sequential (or spatially local) it is, and bounded concurrency requires a parameter (D) to limit the number of outstanding scheduled requests. The best values for all these parameters depend on the workload characteristics and the load on the system. Since the relationship between workload characteristics and the best parameter values can be complex, and workloads and system loads vary over time, it is impractical for an administrator to provide the values for these parameters. We implemented an automated, adaptive method to set the per-application variable batch sizes and the concurrency parameters.

Adapting batch sizes: As we showed in section 4.2, increasing the batch size for application workloads that are sequential or spatially local improves the efficiency of the storage server by reducing the disk seeks, at some cost to the fairness. Ideally, one would set the batch size large enough to capture the sequentiality of each workload, but no larger. We do this by periodically setting the batch size of the application to its average recent *run length* (up to a maximum value). A run is a maximal sequence of requests from a workload that are within a threshold distance of the previous request — we used a threshold distance of 128KB, which is tunable. Algorithm 1 shows the pseudo-code that tracks the last K run lengths; the average recent K run lengths used as batch size L_i.

Adapting concurrency: As discussed in section 4, the efficiency of the storage server generally increases as the concurrency of the server is increased; however, too large an output queue may lead to a loss in fairness. The length of the output queue required to maintain proportional service depends not only on the weights of the applications but also on the number of pending requests. For example, consider two closed applications with 16 IOs pending at all times and weights in the ratio 1:4. Now, in the output queue of length D, we should have $D/5$ requests from a_1 and $4D/5$ requests from a_2. When D is larger than 20, all 16 pending requests of a_2 are in the output queue, and it does not have any more requests to send; the remaining slots in the queue may be occupied by pending requests from a_1 (which still has 12 pending requests in the DRR queue) affecting the fairness guarantees. This is because DRR can only guarantee proportional service so long as the applications are backlogged — that is, there are enough pending requests in each application queue to use up the tokens available and fill the output queue. Thus, we need to adapt the length of the output queue based on the number of requests pending from an application and its share.

```
1  LT = 128K (locality threshold);
2  int runCount[K], runPos[K];
3  int current = 0;
4  Compute Locality()
5  // If request address is not within threshold,
   start new run;
6  reqLBN = logical block number of current
   request;
7  if ( |runPos[current] - reqLBN| > LT) then
8      current++;
9      if (current == K) then
10         current = 0
11     end
12     runCount[current] = 0;
13 end
14 runCount[current]++;
15 runPos[current] = reqLBN;
16 Add request to corresponding DRR queue;
17 Periodically: (every 1 second)
18 L_i = average of non-zero runCount[] entries;
```

1: Calculating average run length

```
On Request Arrival:
  Compute Locality();
  Enqueue request in application's queue;
  Dequeue request();
On Request Completion:
  D = D -1;
  Dequeue request();
```

2: Adaptive DRR algorithm

```
1  DC_i: deficit count of application a_i;
2  P_i: number of requests pending in output queue
   Q_D;
3  R_i: number of requests pending in application
   queue Q_i;
4  curHead = index of current queue;
5  Dequeue Request():
6  for count ← 1to N do
7   i = curHead;
8    // If inactive, be work conserving and go to
   next queue
9    if ( P_i + R_i == 0) then
10       curHead++;
11       if (curHead == N) then
12           curHead = 0
13       continue;
14   // If active and has request, send it
15   if ( DC_i ≥ 1 AND R_i > 0) then
16       DC_i = DC_i-1;
17       R_i = R_i - 1;
18       P_i = P_i + 1;
19       D = D + 1;
20       Send request from a_i;
21       return;
22   // If active with no request, return
23   if ( P_i > 0 AND R_i == 0) then
24       return // Do not send more;
25   curHead++;
26   if (curHead == N) then
27      curHead = 0
28 // Deficit count is zero, replenish and start over
29 for i ← 1 to N do
30   if (a_i deserves quantum) then
31       DC_i = G_i
32 goto line 6;
```

3: DRR request dispatching.

A method to control the concurrency to maximize efficiency while maintaining fairness is shown in Algorithm 3. In order to maximize the efficiency of the server, we allow the concurrency to increase so long as each active application that has tokens for a round has pending IOs in its DRR queue. If the current application a_i has no pending requests in the DRR queue we stop sending requests (thereby decreasing concurrency as requests complete at the server) until one of two events occurs: either a_i sends a new request (perhaps triggered by the completion of an earlier request) or it completes all its requests in the output queue. In the first case, we continue adding a_i's requests to the output queue. In the second case, we declare a_i inactive and continue serving requests from the next DRR queue. In addition, when an application runs out of tokens, the round continues with the next DRR queue. An application is considered active if it has at least one request in the scheduler input queue, output queue, or outstanding at the server. Since every active application receives the full service it is entitled to in each round, the algorithm guarantees proportional service for all active applications. Finally, Algorithm 2 shows the overall adaptive DRR algorithm.

5 Analytical Bounds

Increasing the concurrency and the per-application batch sizes for sequential or local workloads improves the efficiency of the fair scheduler, but at some cost in fairness, as we have observed. In this section, we present some analytical bounds on how far the resulting scheduler can deviate from proportional service.

Most fair schedulers such as WFQ [5], SFQ [8], Self-Clocked [6] and DRR [18], guarantee that the difference between the (weight-adjusted) amount of service obtained by any two backlogged applications in an interval is bounded. The bound is generally independent of the length of the interval. During any time interval $[t_1, t_2]$, where two flows (applications) f and g are backlogged for the entire interval, the difference in aggregate cost of requests completed for f and g, is given by:

$$\left| \frac{S_f(t_1,t_2)}{w_f} - \frac{S_g(t_1,t_2)}{w_g} \right| \leq \frac{c_f^{max}}{w_f} + \frac{c_g^{max}}{w_g} \qquad (3)$$

where c_i^{max} is the maximum cost of a request from flow a_i [6, 8]. Cost is any specified positive function of the requests; for example, if the cost of each request is one, the aggregate cost is the number of requests. A similar (but weaker) bound has been shown for the basic DRR algorithm [18].

When the server is allowed to have multiple outstanding requests simultaneously, the bound is larger. For example, Jin et al. [12] show that in SFQ(D), where the server has up to D outstanding requests, the bound in Eq. 3 is multiplied by $(D+1)$. In our case, as shown below, the bound grows as both D and the maximum value of the batch sizes.

Theorem 1. *During any time interval $[t_1, t_2]$, where two applications a_i and a_j are backlogged, the difference in weight-adjusted amount of work completed by DRR using corresponding batch-sizes G_i, G_j, and concurrency D is bounded by:*

$$\left| \frac{S_i(t_1,t_2)}{w_i} - \frac{S_j(t_1,t_2)}{w_j} \right| \leq 2\left(\frac{G_i}{w_i} + \frac{G_j}{w_j}\right) + D\left(\frac{1}{w_i} + \frac{1}{w_j}\right)$$

Essentially, the theorem says that the bound on unfairness increases proportionally with a linear combination of the concurrency bound D and the batch size parameters G_i and G_j. We present a proof for this theorem in the Appendix.

Parameters D,[G_1, G_2, G_3]	r_1 **(MB/s)**	r_2 **(MB/s)**	r_3 **(MB/s)**	$\mathscr{E}$
1,[1,3,5]	0.52	1.55	2.59	0.53
8,[1,3,5]	0.84	2.51	4.18	0.86
16,[1,3,5]	**0.97**	**2.91**	**4.84**	**0.99**
8,[8,24,40]	0.85	2.53	4.22	0.86
8,[16,48,80]	0.84	2.49	4.19	0.85
8,[32,96,160]	0.85	2.51	4.22	0.86

(a) Workload RRR: stand alone throughput is R:8.8MB/s

Parameters D,[G_1, G_2, G_3]	r_1 **(MB/s)**	r_2 **(MB/s)**	r_3 **(MB/s)**	$\mathscr{E}$
1,[1,3,5]	1.27	3.78	6.3	0.39
8,[1,3,5]	1.61	4.83	8.04	0.49
16,[1,3,5]	2.12	6.34	10.43	0.64
8,[8,24,40]	2.26	6.76	11.3	0.69
8,[16,48,80]	2.46	7.33	12.33	0.75
8,[32,96,160]	2.78	2.51	13.96	0.85
8,[16,96,240]	2.91	8.69	14.62	0.89
8,[16,128,320]	**2.98**	**8.81**	**14.95**	**0.91**

(b) Workload RLL: stand alone throughputs are: R:8.8MB/s, L:41.85M

Parameters D,[G_1, G_2, G_3]	r_1 **(MB/s)**	r_2 **(MB/s)**	r_3 **(MB/s)**	$\mathscr{E}$
1,[1,3,5]	1.87	5.58	9.49	0.4
8,[1,3,5]	1.74	5.16	8.76	0.37
16,[1,3,5]	2.45	7.32	12.4	0.53
8,[8,24,40]	2.94	8.77	14.91	0.64
8,[16,48,80]	3.62	10.79	18.44	0.78
8,[32,96,160]	4.21	12.51	21.38	0.91
8,[16,96,240]	4.11	12.24	20.92	0.89
8,[16,128,320]	**4.69**	**14.08**	**23.83**	**1.02**

(c) Workload LLL: stand alone throughput is L:41.85MB/s.

Parameters D,[G_1, G_2, G_3]	r_1 **(MB/s)**	r_2 **(MB/s)**	r_3 **(MB/s)**	$\mathscr{E}$
1,[1,3,5]	1.09	3.26	5.43	0.13
8, [1,3,5]	2.28	6.79	11.32	0.26
16,[1,3,5]	3.22	9.61	15.39	0.36
8,[8,24,40]	5.03	15.05	25.06	0.58
8,[16,48,80]	5.92	17.71	29.63	0.68
8,[32,96,160]	6.22	18.59	31.21	0.72
8,[128,384,640]	7.06	21.12	35.86	0.82
8,[256,768,1280]	**8.03**	**24.02**	**40.79**	**0.94**

(d) Workload SSS: stand alone throughput is S:77.8MB/s.

Table 2: Measured throughput and efficiency for various settings of concurrency bound and batch size.

6 Experimental Evaluation

In this section, we evaluate our mechanisms for improving the I/O efficiency of proportional share schedulers. We used a variety of synthetic workloads and trace replay workloads in our experiments. Our results are based on the modified DRR scheduler, but our techniques are general enough that they can be applied to other proportional share schedulers.

Overall, we highlight two main points in our evaluation. First, we show how the two parameters we introduced, bounded concurrency, and the variable batch size, can be adjusted to get high efficiency without a significant degradation in fairness. We don't use adaptation, but instead use fixed values of these parameters to study their affect. Since our approach trades off short term fairness in order to get higher I/O efficiency, we evaluate both fairness and efficiency. Second, we show how these parameters can be adapted for dynamically changing workloads. We also compare our adaptive DRR mechanism with other existing algorithms such as anticipatory scheduler, base DRR and SFQ(D) [12].

6.1 Experimental Setup

Our experimental setup consists of a Linux kernel module that implements our mechanisms in a modified DRR scheduler. The module creates a bunch of pseudo devices (entries in /dev), which are backed up by a block device that can be a single disk, a RAID device or a logical volume. Different applications access different pseudo devices. This allows us to classify requests from different applications, and we can set weights for each pseudo device. Our module intercepts the requests made to the pseudo devices and passes them to the lower level Anticipatory scheduler in Linux based on the DRR algorithm with our modifications. Anticipatory scheduler then dispatches these requests based on its own seek minimization algorithm, we don't make any modifications to it.

We use a variety of synthetic micro-benchmarks and trace-replay workloads in our experiments. We experimented with three synthetic workloads and four different workload mixes. The random workload **R** represents an application with 16 pending IOs of 32KB each distributed randomly over the volume. The throughput of this random workload when running in isolation is 8.8MB/s (281 IOPS). The spatially local workload L does 32K sized IOs separated by 16K each. This highly local application has throughput, running in isolation, of 41.85 MB/s (1339 IOPS). The sequential workload sends 32K sized sequential IOs and has overall throughput of 77.8 MB/s (2490 IOPS) in isolation. We consider 4 different mixes representing different number of

random, local, and sequential workloads, defined as as RRR, LLL, SSS and RLL. Here RLL represents one random and two local workloads. The weights are assigned in ratio **1:3:5** in all cases.

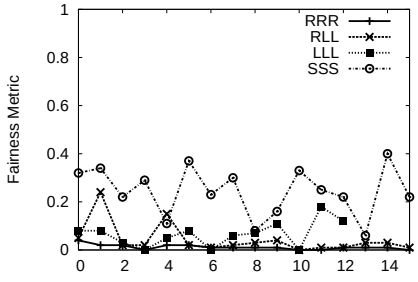


Figure 4: Fairness index $\mathscr{F}$ over time, for one second measurement intervals. For each workload combination, the parameter values with highest efficiency were used.

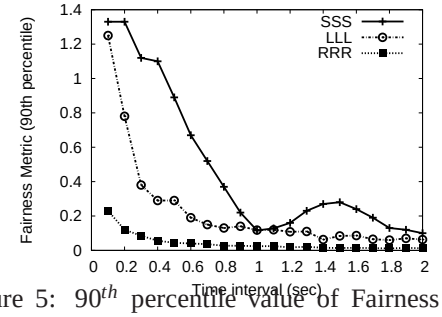


Figure 5: 90^{th} percentile value of Fairness index $\mathscr{F}$ for various measurement intervals over which fairness is computed. For each workload set, the parameter combination with the best efficiency is used.

6.2 I/O Efficiency

In section 4, we showed the impact of individual parameters on fairness and I/O efficiency based on micro-benchmarks. In this section, we look at the combined effect of all the parameters. Our goal is to show that we can adjust these parameters to obtain high I/O efficiency. Table 2(a),(b),(c) and (d) show the measured throughput and efficiency metrics for different parameter values, of workload mixes RRR, RLL, LLL and SSS respectively. These results show that the baseline DRR scheduler (where $D = 1$ and $G = 1$) does indeed exhibit poor I/O efficiency, between 0.13 (for the SSS workload) and 0.53 (for the RRR workload). Our mechanisms improve I/O efficiency to the levels above 90%, improving the performance of the baseline DRR scheduler by a factor of two to seven for different workload mixes. Our results indicate the following: (1) The random workload mix (RRR) is unaffected by batching parameters and its efficiency is solely dependent on the bounded concurrency (D). (2) Batching helps workloads with locality

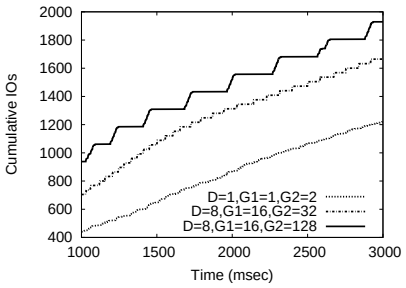


Figure 6: Cumulative IOs of workload with high locality for various values of parameters, D and G. The plot shows the cumulative IOs from 1 to 3 sec. This indicates that the fairness granularity increases with these parameters.

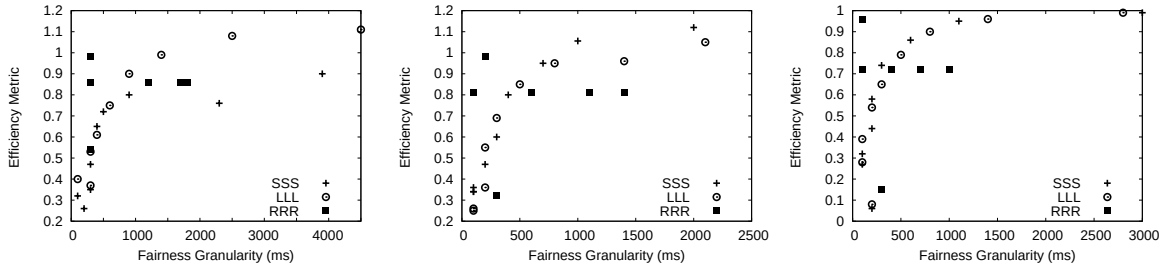
and their performance improves as we increase the batch size. (3) It is possible to get high efficiency with small values of D. This is important, since we have already shown that setting D to a large value causes fairness to deteriorate significantly.

Figure 4 shows the corresponding fairness for one second intervals using the parameter settings that provides the highest I/O efficiency for each workload (i.e., the rows in bold face). Though the fairness is below 0.1 for most workloads at one second granularity, there are cases where the parameter settings corresponding to the highest I/O efficiency lead to poor fairness (e.g., up to 0.4 for the SSS workload). We note that the baseline DRR scheduler has perfect fairness becasue it uses $D = G = 1$.

6.3 Fairness Granularity

We have shown earlier that the fairness index $\mathscr{F}$ depends on the time interval over which it is computed. Also the analysis shows that the worst case fairness bound increases with increase in parameter values D and G, and so does the fairness granularity. In this section we show how the value of $\mathscr{F}$ changes with respect to the time interval over which it is computed.

For each of the workload mixes RRR, LLL, and SSS, we computed the fairness index values as a function of the measurement time interval t. That is, we computed $\mathscr{F}(0,t)$, $\mathscr{F}(t,2t)$, $\mathscr{F}(2t,3t)$, ... Figure 5 shows the 90^{th} percentile of this set for values of t ranging from 100ms to 2000ms. For each workload mix, we used the parameter combination that gave the best efficiency: $D = 16$ and small values of batch size for RRR, and $D = 8$ and large values of batch size for the LLL and SSS workload mixes. The RRR workload has good fairness $\mathscr{F}$ (< 0.1) for measurement intervals of 300ms or higher, whereas the other workloads require 1 second or more to achieve low fairness values. While the fairness generally improves with higher measurement intervals, the



(a) 1-disk (b) striped, 2-disks (c) striped, 4-disks

Figure 7: Efficiency metric $\mathscr{E}$ with various time intervals over which fairness is very good (< 0.1) for three different workload mixes.

changes are not monotonic. For the SSS workload, the algorithm gains efficiency by allocating each workload a large batch in one round, and then allocating no service to it for several rounds. An interaction between the high batch size and the measurement interval cause a bump in the fairness graph, since one measurement interval may have more rounds with large batches allocated than the next. As such, the proportion of service received by a workload may be too high in one measurement interval, and too low in the next. However, the effect declines as the measurement interval grows larger. Overall, the fairness granularity is larger for the SSS case than for the other workload mixes.

These results are also in agreement with our analysis, which shows that the worst case fairness bound increases in proportion to sum of the queue length and batching parameters. To illustrate this further, we experimented with two workloads, one random and one local, with weights set in the ratio 1:2. Figure 6 shows the cumulative IOs completed for the local workload with increasing values of the two scheduler parameters. It shows that higher values for parameter settings result in bigger steps and bursts. Thus, if we measure throughput over short periods, it is quite variable and the fairness can be poor. If fairness is measured over longer periods, the throughput smooths out, and the fairness is good.

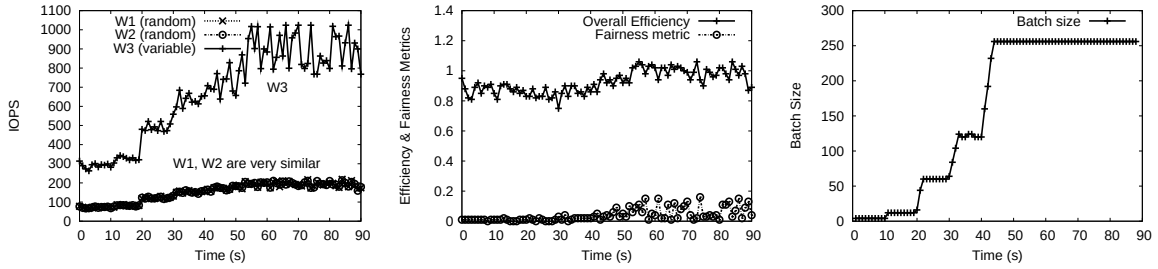
6.4 Efficiency and Fairness Granularity

In this section we look at the relationship between *fairness granularity* and *efficiency*. For this experiment, we assume that the user needs very good fairness, say, a fairness index $\mathscr{F}$ less than 0.1. Figure 7 shows how the efficiency of the scheduler varies with the fairness granularity. As before, the workload weights are 1:3:5. Each point represents one parameter setting for one workload mix in one storage configuration, and the efficiency is plotted against the fairness granularity $\delta(0.1)$. We evaluated three different back-end devices - one disk, two and four disk striped LVMs. The parameter settings are not shown (to avoid cluttering the figures), but we note the parameter settings for some interesting points below. In these plots, the ideal scheduler would be in the top left-hand corner — high efficiency combined with a low fairness granularity. For the random workload mix (RRR), the best combination of efficiency and fairness is achieved at a low fairness granularity (300ms or less); the corresponding parameter settings are D=16 and G=[1,3,5] in all configurations. Higher batch sizes for the RRR workload mix increase the fairness granularity without any improvement in efficiency. For the workloads with significant locality or sequentiality, the efficiency increases with the fairness granularity. In the case of the LLL workload mix, 90% efficiency is achieved at a fairness granularity of 800–900ms; this corresponds to the parameter setting $D = 8$, $G = [64, 192, 320]$ in all three configurations. The third workload mix, SSS, is the most difficult test of the scheduler, because it is hard to retain efficiency when mixing sequential workloads. In this case, 90% efficiency is achieved at a fairness granularity of 3900ms for the single disk configuration, using the parameter setting $D = 8$, $G = [256, 768, 1280]$. On the striped volume configurations, 90% efficiency is achieved for the SSS workload mix at a fairness granularity of 700–1100ms (Figures 7(b) and 7(c)). *Overall, we conclude that fairness granularity can be traded for efficiency in a proportional share I/O scheduler.*

6.5 Adapting parameters to workloads

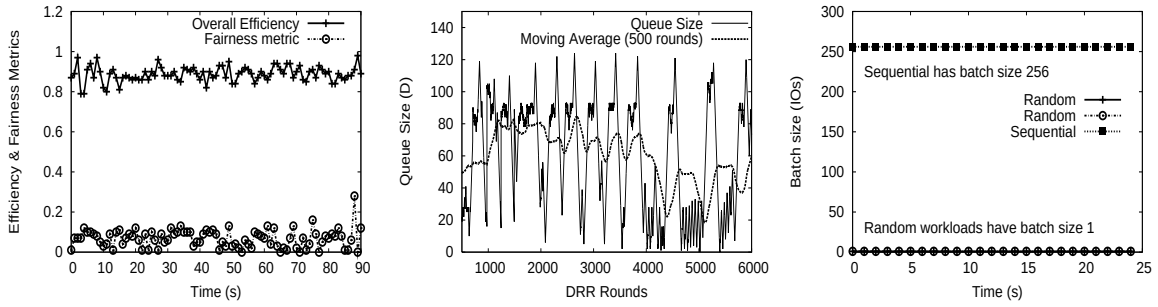
We have so far presented results with fixed values of the concurrency and batch-size parameters. We now evaluate the adaptive DRR algorithm presented in Section 4.3.

In our first experiment, we use a mixture of three workloads, initially all random, and let one of the workloads increase its run length every 10 seconds, turning into a more sequential workload. Ideally, as the third workload gets more sequential, its batch size needs to be adjusted to reflect this change. The weights of the workloads are assigned in ratio 1:1:4, and each workload issues IOs of 32KB on a 2-disk stripe. Figure 8(a) shows the overall throughput with the adaptive DRR algorithm increases over time as one of the workloads be-



(a) Throughput (b) I/O Efficiency and Fairness (c) Batch size

Figure 8: Dynamically adapting batch size as one of the workloads becomes more sequential over time, increasing its run length every 10 seconds.



(a) I/O efficiency and Fairness (b) Number of outstanding requests (D) (c) Batch size

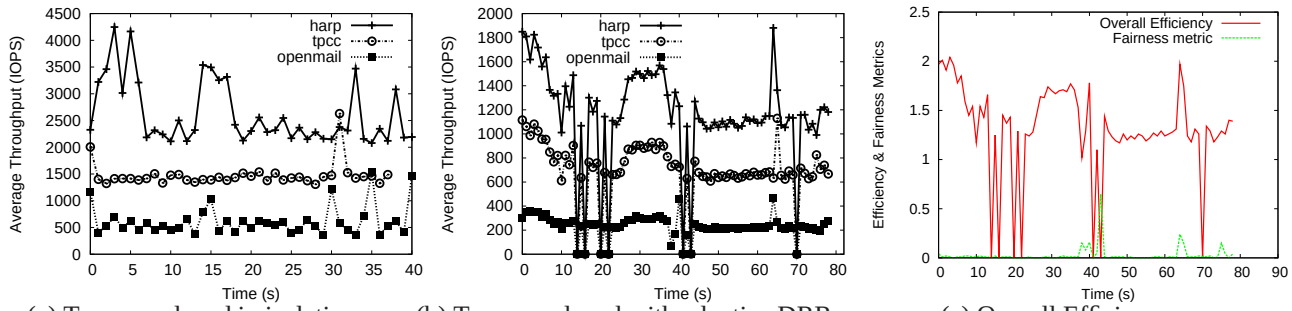
Figure 9: Dynamically adapting queue length as one of the workloads decreases its concurrency from 128 to 4 at 10 seconds granularity.

comes more sequential. We also plot the efficiency and fairness (with 1 second measurement intervals) for the same experiment in Figure 8(b) and the batch size of the workload which changes its run length during the experiment in Figure 8(c). These results show that the adaptive DRR is able to keep high I/O efficiency and trades off short term fairness by letting the fairness index to increase up to 0.1. It achieves this by varying the batch size for the changing workload as it increases its run-length as shown in Figure 8. We also sampled the queue size at the storage system every second. Both the mean and median queue length was 24.

In our second experiment, we again consider a mixture of three workloads, two random and one sequential, and let the sequential workload vary its concurrency (the number of requests it has outstanding) from 128 to 4 at 10 second intervals. The random workloads each have a fixed concurrency of 32 and issue 32KB IOs. Since the sequentiality characteristics of the workloads do not vary, the algorithm keeps the batch sizes for the workloads unchanged throughout — 256 for the sequential workload and 1 for the random workload, as shown in Figure 9(c). The overall concurrency — the total number of outstanding requests — decreases from 196 to 68 over a period of 150 seconds. To adapt to the changing concurrency of the workload, the algorithm automatically adjusts the number of requests at the back-end queue, as shown in Figure 9(b). As the pending count for the sequential workload decreases, so does the average queue length. However, the sequential workload gets a large batch of size 256 (because it is sequential) and then misses its turn for the next 64 rounds (because its weight is 4). During those rounds, the queue size is high because of the backlog from the random workloads. The large back-end queue allows for good seek-optimization and high efficiency with random requests. Figure 9(a) shows the efficiency and fairness for the duration of the experiment. The overall efficiency is close to 90% and fairness measured over one second intervals is around 0.1, which indicates that the adaptive algorithm successfully manages the back-end queue depth to obtain good efficiency and fairness despite the rapidly changing workload.

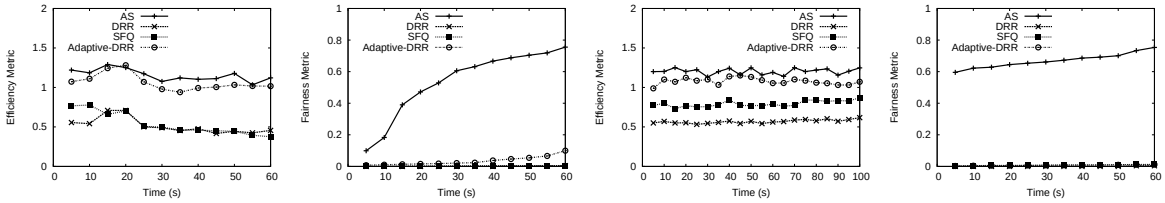
6.6 Comparison With Other Approaches

In this section, we compare the performance of our adaptive DRR mechanism with some of the well known algorithms, that are used in practice and are proposed in literature. We compare with three other algorithms: Anticipatory scheduler, SFQ(D) and base DRR. Anticipatory scheduler is available in Linux distribution and other two we implemented as modules. First we use the workload



(a) Traces replayed in isolation (b) Traces replayed with adaptive DRR (c) Overall Efficiency

Figure 10: Running three different traces (openmail, tpcc and harp) using adaptive DRR.



(a) Efficiency metric comparison (b) Fairness index comparison (c) Efficiency metric comparison (d) Fairness index comparison

Figure 11: Comparison of adaptive DRR with anticipatory, SFQ(D) and base DRR schedulers.

mix of three workloads, which are all random and we make one of them increase the run length every 10 seconds, going from 1 to 1024. The weights are set to 1:1:4 and all workloads are doing 16 concurrent IOs of 32KB each. Figure 11a and 11b show the efficiency and fairness index for various schedulers. Note that adaptive DRR is very close to anticiapatory scheduler in terms of efficiency. Other mechanisms such as base DRR and SFQ(D) show poor efficiency and that gets worse as batch size increases. This is mainly because they can't adapt to the change in batch size. However, these mechanisms provide very good fairness. Note that anticipatory scheduler provides very poor fairness, where as adaptive DRR is able to provide very good fairness close to the SFQ(D) and base DRR, even for a batch size of 1024.

Next we experimented with variable concurrency workload, where third workload changes its concurrency from 128 to 4 by periodically decreasing the number of outstanding IOs every 10 seconds. Figure 11c and 11c show the efficiency and fairness index for various schedulers. Again note that adaptive DRR is close to anticiapatory scheduler in terms of efficiency. However anticipatory scheduler provides very poor fairness, where as adaptive DRR is able to provide very good fairness close to the SFQ(D) and base DRR. *These results show that adaptive DRR is able to provide good fairness along with providing high efficiency even in presence of workload changes.*

6.7 Postmark Experiments

6.8 Experiments with Traces

In this section, we experiment with real world traces to evaluate our adaptive scheduler. We used three representative traces for mail server (*openmail*), data base (*tpcc*), and file system (*harp*) workloads. We replayed these traces on a 4-disk logical volume [1]. Figure 10(a) shows the throughput obtained by traces when they are run separately, in isolation. Since traces are open workloads, the rate of request completion is also bounded by the actual arrivals in the trace. We observe that on average openmail, tpcc and harp get 540, 1470 and 2800 IOs respectively. Then we ran these traces using DRR with weights in ratio 1:3:5. Figure 10(b) shows the throughput while running all three simultaneously. Note that individual IO throughputs are lower than those obtained in isolation because system cannot provide the full desired service to all of them. Figure 10(c) shows the overall efficiency of the system (this calculation is done assuming a steady state average throughput in isolation). The efficiency is around 1.4 due to two reasons: (1) combining multiple traces leads to an increase in system utilization as the overall arrival rate increases, and (2) combining workloads causes the size of the I/O queues to increase, providing more opportunities for the lower level schedulers to improve the efficiency. These results show that our adaptive DRR algorithm handles the substantial variation in workload characteristics exhibited by real world

workloads.

7 Conclusions

In this paper we studied the trade-off between fairness and efficiency in a shared storage server. We showed how this trade-off can be controlled using two parameters: variable size batching and the depth of the scheduler's output queue. We highlight the important characteristics of each of these parameters and show that they can be tuned to trade off fairness granularity — short term fairness — with efficiency. We then present a self-tuning algorithm that sets the values of these two parameters based on dynamic workload characteristics. We validated our approach by an extensive experimental study using both synthetic micro-benchmarks and actual traces. The approach is also backed up by a formal framework and analysis that supports the experimental results. Experimental results using a variety of workload mixes indicate that an I/O efficiency of over 90% is achievable by allowing the scheduler to deviate from proportional service for a few seconds at a time.

References

[1] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan. Buttress: A toolkit for flexible and high fidelity I/O benchmarking. In *Proc. of Conf. on File and Storage Technologies (FAST'04)*, pages 45–58, March 2004.

[2] J. C. R. Bennett and H. Zhang. WF^2Q: Worst-case fair weighted fair queueing. In *Proc. of INFOCOM '96*, pages 120–128, March 1996.

[3] J. Bruno, J. Brustoloni, E. Gabber, B. Ozden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *Proc. of the IEEE Int'l Conf. on Multimedia Computing and Systems, Volume 2*. IEEE Computer Society, 1999.

[4] D. D. Chambliss, G. A. Alvarez, P. Pandey, D. Jadav, J. Xu, R. Menon, and T. P. Lee. Performance virtualization for large-scale storage systems. In *Symposium on Reliable Distributed Systems*, pages 109–118, Oct 2003.

[5] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queuing algorithm. *Journal of Internetworking Research and Experience*, 1(1):3–26, September 1990.

[6] S. Golestani. A self-clocked fair queueing scheme for broadband applications. In *Proc. of INFOCOM'94*, pages 636–646, April 1994.

[7] P. Goyal, X. Guo, and H. M. Vin. A hierarchial cpu scheduler for multimedia operating systems. *SIGOPS Oper. Syst. Rev.*, 30(SI):107–121, 1996.

[8] P. Goyal, H. M. Vin, and H. Cheng. Start-time fair queuing: A scheduling algorithm for integrated services packet switching networks. Technical Report CS-TR-96-02, UT Austin, January 1996.

[9] A. G. Greenberg and N. Madras. How fair is fair queuing. *J. ACM*, 39(3):568–598, 1992.

[10] A. Gulati, A. Merchant, and P. Varman. *p*Clock: An arrival curve based approach for QoS in shared storage systems. In *Proc. of ACM SIGMETRICS*, pages 13–24, June 2007.

[11] L. Huang, G. Peng, and T. cker Chiueh. Multi-dimensional storage virtualization. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 14–24, June 2004.

[12] W. Jin, J. S. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *SIGMETRICS '04/Performance '04: Proceedings of the joint international conference on Measurement and modeling of computer systems*, pages 37–48, June 2004.

[13] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *Trans. Storage*, 1(4):457–480, 2005.

[14] C. Lumb, A. Merchant, and G. Alvarez. Façade: Virtual storage devices with performance guarantees. *Proc of Conf. on File and Storage Technologies (FAST'03)*, pages 131–144, March 2003.

[15] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn. Efficient guaranteed disk request scheduling with fahrrad. *SIGOPS Oper. Syst. Rev.*, 42(4):13–25, 2008.

[16] A. L. N. Reddy and J. Wyllie. IO issues in a multimedia system. *IEEE Computer*, 27(3):69–74, 1994.

[17] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *Proc. of ACM SIGMETRICS*, pages 44–55, June 1998.

[18] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *Proc. of SIGCOMM '95*, pages 231–242, New York, NY, USA, August 1995. ACM Press.

[19] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: performance insulation for shared storage servers. In *Proc. of Conf. on File and Storage Technologies (FAST'07)*, pages 5–5, 2007.

[20] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[21] T. M. Wong, R. A. Golding, C. Lin, and R. A. Becker-Szendy. Zygaria: Storage performance as managed resource. In *Proc. of RTAS*, pages 125–34, April 2006.

[22] J. C. Wu and S. A. Brandt. The design and implementation of Aqua: an adaptive quality of service aware object-based storage device. In *Proc. of IEEE/NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2006)*, pages 209–18, May 2006.

[23] J. Zhang, A. Sivasubramaniam, Q. Wang, A. Riska, and E. Riedel. Storage performance virtualization via throughput and latency control. In *Proc. of MASCOTS*, pages 135–142, September 2005.

8 Appendix

8.1 Analysis

Theorem 2. *During any time interval $[t_1,t_2]$, where two applications a_i and a_j are backlogged, the difference in weight-adjusted amount of work completed by DRR using corresponding batch-sizes G_i, G_j, and concurrency D is bounded by:*

$$\left|\frac{S_i(t_1,t_2)}{w_i}-\frac{S_j(t_1,t_2)}{w_j}\right| \leq 2(\frac{G_i}{w_i}+\frac{G_j}{w_j})+D(\frac{1}{w_i}+\frac{1}{w_j})$$

Proof. Consider an interval $[t_1,t_2]$ where application a_i gets m_i non-zero quantum allocations. Each quantum allocation corresponds to batch size G_i of a_i. The total amount of service obtained by a_i can be written as:

$$S_i(t_1,t_2) = m_iG_i + DC_i(t_1) + d_i(t_1) - DC_i(t_2) - d_i(t_2) \quad (4)$$

Here, $DC_i(t)$ denotes the number of tokens a_i has at time t and $d_i(t)$ denotes the number of outstanding scheduled (but not completed) requests from a_i at time t.

Noting that $0 \leq DC_i(t) \leq G_i$ and $0 \leq d_i(t) \leq D$, we can upper bound the expression for S_i as:

$$S_i(t_1,t_2) \leq m_iG_i + G_i + D \quad (5)$$

Similarly, the lower bound is:

$$S_i(t_1,t_2) \geq m_iG_i - G_i - D \quad (6)$$

Considering the upper and lower bounds for applications a_i and a_j respectively, we get:

$$\frac{S_i(t_1,t_2)}{w_i} \leq \frac{m_iG_i}{w_i} + G_i/w_i + D/w_i \quad (7)$$

$$\frac{S_j(t_1,t_2)}{w_j} \geq \frac{m_jG_j}{w_j} - G_j/w_j - D/w_j \quad (8)$$

Hence the difference is bounded by:
$\frac{S_i(t_1,t_2)}{w_i} - \frac{S_j(t_1,t_2)}{w_j} \leq \frac{m_iG_i}{w_i} + \frac{(G_i+D)}{w_i} - \frac{m_jG_j}{w_j} - \frac{(G_j+D)}{w_j}$

Let τ_i and τ_j be the number of rounds between successive quantum allocations to applications a_i and a_j respectively. These values will non-zero because if an application has high batch size G_i, then it may have to skip a few rounds in order to maintain proportionate fairness over a long run. Figure 12 illustrates the parameters used in proof. Here application a_i gets its quantum allocation G_i every alternate round. Hence τ_i = 2. Also within a time interval $[t_1,t_2]$, a_i may get m_i = 10 such allocations. Similarly application a_j gets its quantum allocation of G_j every fourth round, hence τ_j =4. Also in the same interval a_j will get at least 4 (m_j = 5) allocations. The numbers τ_k and m_k depend on the batch size and weights of different applications.

The length of time interval $[t_1,t_2]$ is at least $(m_i-1)\tau_i$. Consider the other application a_j: during interval $[t_1,t_2]$, it will receive at least m_j quantum allocations given by:

$$m_j = \lfloor (m_i-1)\tau_i/\tau_j \rfloor \quad (9)$$

Based on the computation of G_i and τ_i, we also know that

$$\frac{G_i * \tau_j}{G_j * \tau_i} = \frac{w_i}{w_j} \quad (10)$$

This is because the overall allocation per round must be in ratio of the weights. Substituting m_j and G_j/w_j from the equations above, we get:

$$\begin{aligned} m_jG_j/w_j &\geq G_j((m_i-1)\tau_i/\tau_j - 1)/w_j & (11)\\ &= G_i\tau_j((m_i-1)\tau_i/\tau_j - 1)/(w_i\tau_i) & (12)\\ &= G_im_i/w_i - G_i/w_i - G_i\tau_j/(w_i\tau_i) & (13)\\ &= G_im_i/w_i - G_i/w_i - G_j/w_j & (14)\end{aligned}$$

Substituting in the difference computation, we get:
$\frac{S_i(t_1,t_2)}{w_i} - \frac{S_j(t_1,t_2)}{w_j} \leq \frac{(G_i+D)}{w_i} + \frac{G_i}{w_i} + \frac{G_j}{w_j} + \frac{(G_j+D)}{w_j}$

By grouping the terms for G and D we get:
$\left|\frac{S_i(t_1,t_2)}{w_i} - \frac{S_j(t_1,t_2)}{w_j}\right| \leq 2(\frac{G_i}{w_i}+\frac{G_j}{w_j}) + D(\frac{1}{w_i}+\frac{1}{w_j})$ □

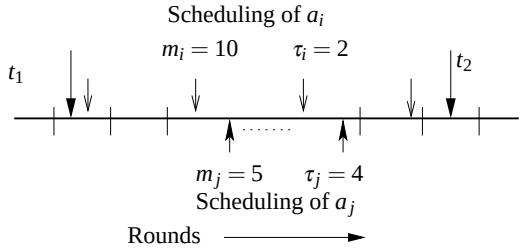
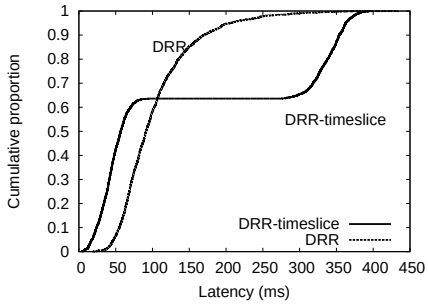


Figure 12: Illustration for proof

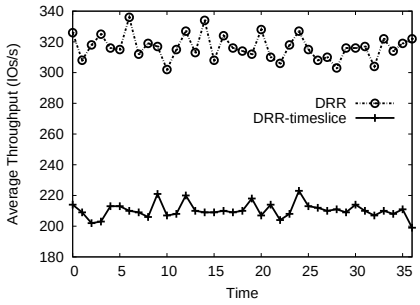
Essentially, the theorem says that the bound on unfairness increases proportionally with a linear combination of the concurrency bound D and the batch size parameters G_i and G_j.

8.2 Time Slicing at Disk

In this section, we take a closer look at the alternative approach of time slicing at the disk and discuss some of the fundamental issues with that approach. We implemented a DRR-timeslice algorithm that does time multiplexing at a fine granularity. The length of an application's time slices is proportional to the weight of the application. If an application has no more requests to send, it will wait



(a) Latency distribution



(b) Throughput

Figure 13: Comparison of time slicing and proportional share scheduling.

if the lower level queue has at least one request pending ($D \geq 1$), otherwise the DRR-timeslice will move on to the next application's time slice. Thus, we chose to end the time slice as soon as an application becomes inactive; we made this choice to make the scheduler work-conserving.

In this experiment, we used four random workloads, each keeping 8 requests pending, with equal weights. The back-end queue depth is 16. We set the time-slice to be 100ms for each workload. Figure 13 shows the cumulative distribution of latency for one of the workloads and the average total throughput. This shows that almost 60% of IOs have a small latency of around 50ms and the remaining have a latency of more than 300ms. This number is dependent on the workloads (four in this case); with a larger number of workloads, the maximum latency would be higher. By contrast, the DRR algorithm has less jitter. DRR also has better overall throughput. DRR obtains around 320 IOs/s, whereas DRR-timeslice obtains only around 215 IOs/s. In the case of time slicing we can only use the concurrency from a single workload (8 in our case), whereas the DRR algorithm maintains 16 IOs in the back-end queue. Thus, DRR-timeslice loses the improvements in efficiency associated with higher concurrency (better seek optimizations and higher parallelism). A good time-slicing mechanism should figure out when to isolate and merge the IO workloads, instead of doing strict time multiplexing at all times.