Reprinted from the

Proceedings of the Linux Symposium

July 23rd–26th, 2008
Ottawa, Ontario
Canada

IO Containment

Naveen Gupta
Google Inc.
ngupta@google.com

Abstract

In existing Linux IO schedulers there is no way to differentiate latency-sensitive IO from the background IO. This means that jobs which have strong latency requirements cannot coexist with batch jobs. In the past various ad-hoc solutions have been used to limit latencies such as throttling background workload, periodic syncing to handle delayed writes, and limiting the queue length.

With increasing CPU and memory speeds we generally end up having multiple applications on the same machine sharing the disk. In the absence of differential service, it is not possible to run a latency-sensitive application along with a job doing batch updates. We have added *priorities* in the *anticipatory IO scheduler* so that an IO submitted by an application can be treated differently depending on the priority of IO. We are able to show considerable improvement in the latency of *high priority* applications using this mechanism.

1 Introduction

There is an effort within the Linux community to provide isolation for memory and network resources to run multiple jobs independently on a single machine. IO containment has been a difficult problem for various reasons. The non-deterministic nature of storage devices makes any kind of QoS or isolation a harder problem for IO as compared to other computing resources. Variations in an IO stream—like size, placement of data on platter, or seek distance—have played a major role in determining throughput and latency. This inability to quantify the device characteristics leads to inefficiency when performing any kind of isolation. Even the applications sharing these resources have vastly different performance and service quality requirements.

Disk isolation can be done over one or multiple IO characteristics. For workloads needing different service rates—bandwidth scheduling—which may be implemented as a strict bandwidth guarantee or as proportional scheduling, may suffice. Latency bounds are needed for real-time applications sharing their storage resources, while allocation of time slices is needed for shared usage in applications wanting to schedule for non-deterministic things like rotational delay and seek time rather than raw throughput. Resource accounting frameworks may need to limit the number of seeks in a given container. Relative *priority* of a request as compared to other outstanding requests is helpful in separating *latency-sensitive* traffic from *best-effort* traffic. In context of web searches, we need to serve low-latency user traffic independent of background index updates.

The Linux mainline has support for four different IO schedulers [5]. They have mainly aimed towards improving bandwidth by reordering requests, submitting them in batches, and merging outstanding requests. The *deadline* scheduler tries to process all read requests within a specified time period by selecting expired read requests from a *FIFO* list. The *anticipatory* scheduler [14] is a non-work-conserving scheduler which reduces seeks for a sequential workloads. The *CFQ* scheduler allocates requests fairly amongst various processes, while *NO-OP* provides basic merging. Apart from *CFQ*, which allocates larger time slices to higher priority processes, there is no differential service in the current mainline IO schedulers.

Our current work uses *non-work-conserving* scheduling to provide *priority*-based scheduling by building upon the *anticipatory* scheduler. The current implementation of the *anticipatory* scheduler solves the problem of *deceptive idleness* [14] present in applications which issue synchronous reads. It identifies *deceptive idleness* as a condition where due to a short period of computation, the scheduler incorrectly assumes that the last request-issuing process has no further requests and dispatches the requests from another process. These workloads cause seek-optimizing schedulers to multiplex between requests from different processes. Letting the disk remain idle for a short period lets the scheduler select the next nearby request from the same process, thereby increasing disk performance at the cost of a small loss in utilization. In our work we extend this concept to block pending requests from a *low priority* process for a short duration after submitting a request from a *high priority* process. This prevents an incoming *high priority* request from being delayed due to a dispatched *low priority* request. In addition to this a running batch of *low priority* processes can be interrupted on seeing a request of *high priority*.

2 Related work

Jassura et al. [1] have used both *non-work-conserving* and *work-conserving* approaches at the user and kernel levels to provide differentiated levels of service in web content hosting based on priority. Their main metric for quality of service for an HTTP request was latency. In their implementation, a request may be postponed if the load on the system is already high or if the request is of lower priority. Their solution was not restricted to a particular subsystem and was a preliminary step in investigating QoS mechanisms in web servers. In their study, they observed that they had to limit the number of processes to have differentiated performance. An important finding was that a *non-work-conserving* approach is better since in their implementation a *work-conserving* approach could not do much differentiation with a large number of processes. Our approach isolates latency even with multiple antagonists.

Seetarami Seelam et al. [23] proposed that throttling IO streams in high performance computing systems improved I/O performance. In Linux, similar approaches are sometimes used to limit the latency of a *high priority* job while running a throttled background job with decreased utilization. By using a *non-working-conserving* scheduler we have been able to improve isolation without throttling the IO streams.

Isolation based on time slicing has been used in *CFQ* [3], the *eclipse operating system* [6], and *YFQ* [22]. *Argon* [27] defined insulation as reduced interference between workloads, allowing sharing with a bounded loss of efficiency. But its focus was on throughput efficiency and though usually it reduced average response times, it could increase the variation in latency and worst-case response times. Our implementation of *anticipatory priorities* insulates both the average and maximum latency.

Dynamic selection of IO schedulers [21] investigated the possibility of using runtime switching of IO schedulers to *deadline* for providing latency bounds while using *CFQ* for best throughput traffic. This selection was based on feedback from the IO system and the workload. Such a technique would be difficult to optimize for competing workloads while incurring no overhead for scheduler switches.

In order to compensate for the non-deterministic nature of disks, QoS schedulers rely on conservative estimates of bandwidth and latency. Some implementations use weighted fair queuing coupled with admission control. [7, 11, 15, 16, 19] all use a fair queuing/tag-based approach. Even though request stream patterns and the disk characteristics impact the performance of other applications, these studies either ignore or take a conservative estimate. Another approach taken in IO schedulers [19] is to take into account a disk model, but such techniques are not feasible with numerous disk drives, since most often it is not specified by the manufacturer and, models are an approximate representation for today's complex drives. Latency-bound IO in QoS scheduling is orthogonal to our problem, since we want *prioritized* traffic to be isolated. We don't want it to be delayed and scheduled later to satisfy pre-calculated bounds.

The `badger` Project [13] solved a similar problem of improving the latency of transaction synchronous requests while improving the throughput of asynchronous transaction database requests. They realized that the only latency control available for synchronous requests with current schedulers was to limit the number of pending background requests. Since they were focusing on a database, they chose to implement *priorities* in the *deadline* scheduler since it is the most-used IO scheduler for databases [18]. They implemented one *FIFO* per *priority* level where *priority* was determined when the deadline was set and that *low priority IO* would wait for a *high priority* request. This effectively improved the latency of *high priority* requests in their experiments with better throughput than the no-priority, rate-throttling approach. But since *priorities* determined when the deadline was set, the latency was still bound by this deadline. Our approach has no disk-parameter-agnostic bounds and the non-work conserving approach achieves tighter latency bounds [1]. Also, throughput [4] with the *antic-*

ipatory scheduler is inherently more than with *deadline* schedulers.

ABISS [10] is designed to provide guaranteed reading and writing bit-rates to applications, with minimal overhead and low latency. They use a custom *priority* scheduler and have support for *CFQ*. Their solution could use *anticipatory priorities* since it has lower latency for *higher priority* jobs than *CFQ*.

3 Design

The *anticipatory IO scheduler* [18] is a one-way elevator algorithm with limited backward movement. It has *FIFO* expiration times for both reads and writes. These queues are maintained separately and expiration times are tunable. This is similar to the *deadline* IO scheduler implementation for interrupting the elevator sweep. Another feature of the *anticipatory* scheduler is batching, where bunches of read requests or write requests are submitted together. The *anticipatory* scheduler alternates dispatching read and writes batches to the driver. Also, the scheduler anticipates when reads are dispatched to the driver one at a time. At completion, if the next request is close to the previous request or from same process, it is dispatched immediately.

Priority scheduling changes each of these policies to improve response time for a *high priority* process and prevents starvation of a *low priority* one. To change these policies, two data structures are also changed, `sort_list` and `fifo_list`. Changes to these structures are described below as part of policy changes.

Sort queue: For the one-way elevator, we add two queues per *priority* level so that the requests in a given level can be served independently of requests at any other level according to their layout on the disk. Instead of two `sort_lists`, we now have two `sort_list` structs for each level. These lists contain requests in sorted order according to their layout on disk. A backward seek can still occur when choosing between two IO requests where one is behind the elevator's current position, and the other is in front. As before, if the seek distance to the request in the back of the elevator is less than half the seek distance to the request in front of the elevator, then the request in the back can be chosen.

FIFO queue: Instead of one *FIFO* queue for each request direction, we now have two *FIFO* queues for each *priority* level. Only when the requests from a given *priority* level are being served, the read or write from that level expires to interrupt the IO scheduler in its current one-way sweep or read anticipation. This prevents the starvation of requests in a given *priority* level. The expiration times for requests on these lists are tunable using the existing parameters `read_expire` and `write_expire`.

Changes to Anticipatory Core: In the vanilla *anticipatory* scheduler, when a read request completes, the next request is not dispatched to the driver unless it is from same process or a nearby request [18]. *In the priority scheduler, the next request will wait if it is from a process of lower priority, in anticipation of a request from a process of higher priority. When a request with higher priority is submitted by the application, it will break any anticipation happening in the scheduler*. This helps us prioritize a request over a batch of low-priority requests. It still maintains the earlier statistics on *think time* and *mean seek distance* for the process to decide if it is worthwhile to wait for a request. The read anticipation can be disabled using `antic_expire` set to zero.

Changes to Batch submission: In most IO schedulers, requests are served in batches where a batch is a set of requests of one kind (read or write). For a read/write request, the scheduler submits read requests until there are more read/write requests to submit and batch time has not been exceeded. Before scheduling requests for the alternate direction, the scheduler lets all requests from the previous batch complete.

In addition to the time limit imposed on batches by `read_batch_expire` and `write_batch_expire`, the *priority* scheduler limits the number of requests for a given *priority* level using tokens associated with that *priority* level. Instead of alternately serving read and write batches, the *priority scheduler* selects a particular *priority* level and submits requests in one direction as long as there are requests in the chosen direction or there are tokens left for that level. The batching of requests for a given *priority* level maintains the localization per *priority* level and gets fairly good results due to inherent batching in processes in a given *priority* level. As before, the read and write *FIFO* expiration times are checked only when scheduling IO of a batch for the corresponding (read/write) type.

Handling merges: Merging is another property of various IO schedulers. If a request entering the IO scheduler is contiguous with a request that is already on the queue, either to the front or to the back of a request, it is called a front-merge candidate or a back-merge candidate. If the size of new request is less than what can be handled by the hardware, this new request is merged with the one already in queue. With *priorities*, a request is merged with an already existing request if it satisfies the above criteria and also if it is from the same *priority* level as the existing request. Merging of the request with a request from another level should be possible, but we need to see if this has any performance gain. Also, there is an issue of whether we need to change the *priority* band of the merged request, especially when dealing with logs and journalled file systems due to a large number of merges.

Handling writes

Attaching Priority: The *priority* of a request is derived from the current context and uses the infrastructure in the Linux kernel for *CFQ*. Since writes are submitted asynchronously, they get submitted in the context of background threads or the process doing direct reclaim. A prototype implementation was created to attach *priority* to the page struct in submission context. This *priority* is transferred to the request during submission.

Writes and Page cache: While serving cached read traffic, an application doing writes could effectively wipe out the page cache unless it is submitted to the elevator frequently. One of the possible solutions would be to throttle the dirty path with feedback from the elevator regarding available tokens for that *priority* level. A second approach would be to do memory isolation of an antagonist job. Fianlly, one can periodically sync the dirty page cache. For simplicity we adopted this approach to eliminate interference from background high-throughput jobs. We experimented using both periodic dropping of cache and direct IO for background write traffic.

Workload Description: We have attempted to isolate latency, in particular we want to improve the latency of seek-bound (low throughput) search traffic. When this search traffic is served, in the background, the data that is being used to serve this search traffic (index) is refreshed for more up-to-date results. This interferes with foreground search traffic. Most of the foreground traffic is random synchronous reads, some of which is sensitive to the amount of cache in the system. We have synthesized these two workloads to quantify the above

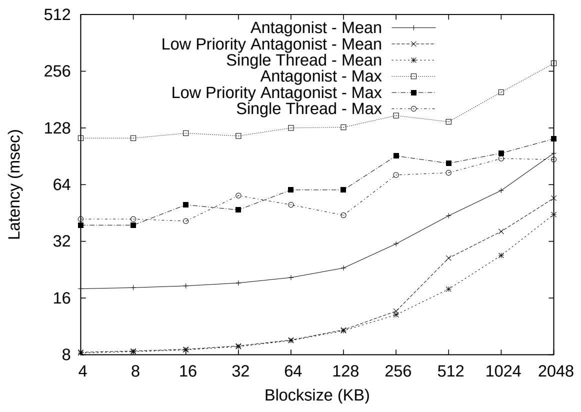

Figure 1: Random read latency – varying block size

problem.

1. Sequential read of 80% of the file followed by random reads to achieve good amounts of hits in undisturbed cache. This is run along with antagonist sequential read/write traffic.

2. Random reads with sequential background read/ write.

4 Experimental Results

We evaluated the performance on two different configurations. The first one is a dual-core 2Ghz system with 8GB of RAM. The test disk is a SATA drive with a capacity of 400G and 8MB onboard cache. The second system is a dual-core 2.4Ghz system with 16GB of RAM and a SATA drive with a capacity of 500G and 8MB onboard cache. Both systems are running a customized 2.6.18 kernel with ext2. The experiments were conducted using synthetic workloads generated using fio [2]. The tests were run on separate drives not running any other workload.

1. ***Random and Sequential Read Latency***

   The first experiment is designed to measure the latency impact of random reads in the presence of competing random reads (Figure 1). It measures the mean and maximum latency of reads for different block sizes. When more than one thread is running, both use the same block size. The three cases in this experiment are

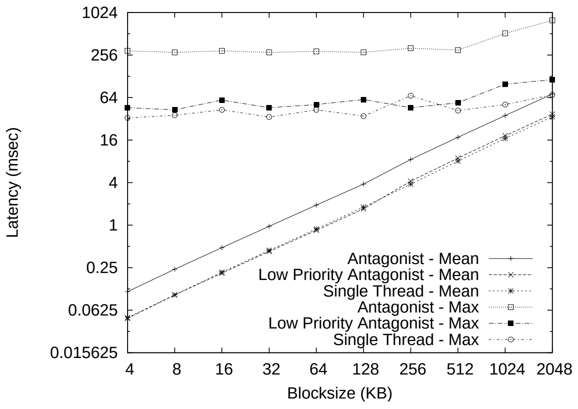

Figure 2: Sequential read latency – varying block size

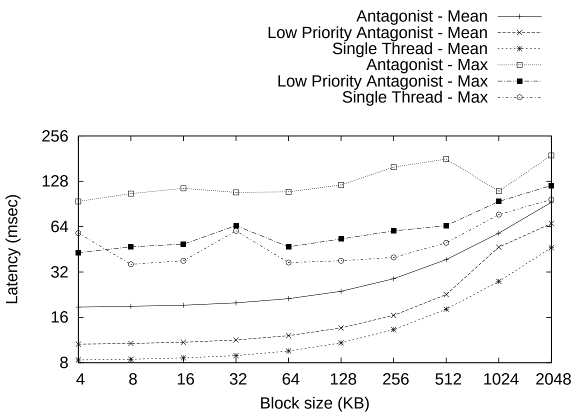

Figure 3: Direct IO latency – varying block size

(a) *Single Thread*: One thread doing random reads.

(b) *Antagonist*: Two threads doing random reads.

(c) *Low Priority Antagonist*: Two threads doing random reads at two different *priority* levels.

When a competing thread is introduced without any *priority (b)*, the average latency is almost two times as compared to when a single thread *(a)* is running and the max latency is more than double. On lowering the IO priority of the antagonist *(c)*, the mean and the maximum latency for the main job is almost similar to when only a single thread is running. Changing the *priority* of the antagonist has isolated latency of foreground job.

For sequential reads, the results are similar. The mean latency and the maximum latency are isolated from competing workload on increasing the IO priority of main thread (Figure 2). In case (b), the maximum latency is more than 4 times, though the average latency is almost twice—like in the case of random reads. Here also, by increasing the *priority* of the main thread (c), the effects of an antagonist are minimal.

2. ***Direct IO***

In the second comparison threads are doing random direct IO (Figure 3). Here also, we have three cases where:

(a) *Single Thread*: One thread running doing direct IO.

(b) *Antagonist*: Two threads doing direct IO at the same *priority* level.

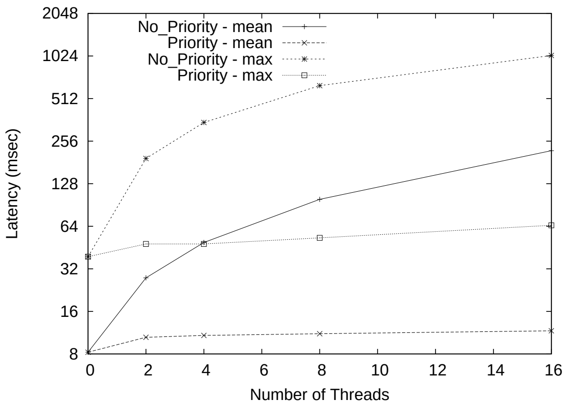

Figure 4: Multiple threads in background – block size 4k

(c) *Low Priority Antagonist*: Two threads doing direct IO at different *priority* levels.

In case (c) the latency is reduced, since the *priority scheduler* anticipates a *high priority* IO even if there is a pending *low priority* IO in queue. This means that a *high priority* IO is not blocked behind a *low priority* even if the submissions happened one after the another. In the case of a *work-conserving* scheduler, a direct IO submission will be handed off to the driver and hence the latency of a *high priority* job is not isolated from *low priority* submissions. For all measured block sizes, the equal *priority* antagonist had more than twice the latency as compared to when a single thread is running, but when *higher priority* was assigned to the main thread, both the average and maximum latency were reduced to a large extent.

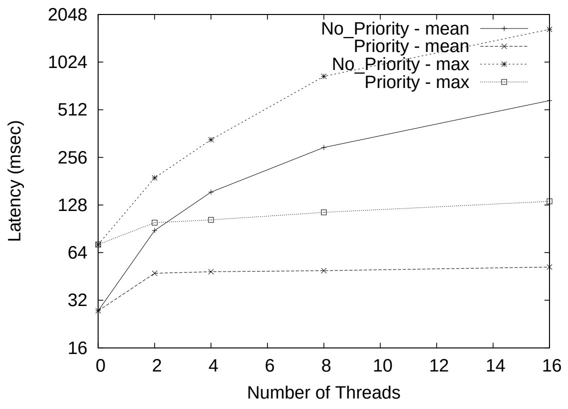

Figure 5: Multiple threads in background – block size 1m

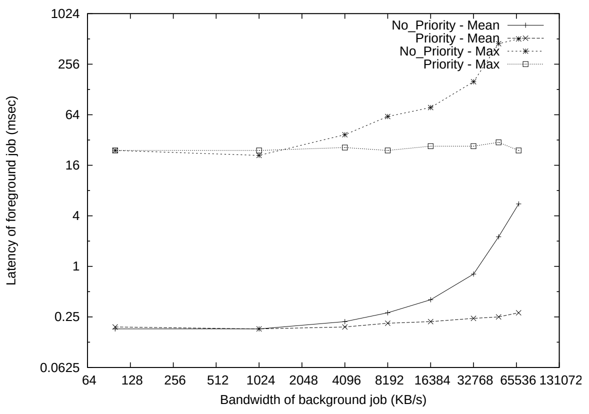

Figure 6: Rate-limited Background Load

3. ***Multiple Background Threads***

   Figure 4 shows the isolation provided to a high priority job in the presence of multiple low priority jobs. It has a thread doing 4k direct IO with a varying number of antagonist jobs doing direct IO of the same block size. When the antagonist threads run at the same *priority* as that of the main thread *(No Priority)*, the latency (both average and maximum) roughly doubles with each doubling of the number of antagonists. On assigning higher priority to the main thread *(Priority)*, the latency is fairly constant even when the number of background threads is increased. Figure 5 plots the results of changing the number of background threads when the block size is 1MB. Irrespective of block size, varying the number of background threads has little or no effect on the latency of the foreground job. Earlier *user-level* and *kernel-level* approaches to using *priorities* in both *work-conserving* and *non-work-conserving* schedulers [1] required restricting the number of jobs to obtain differentiated performance. The current scheme has no such limitation.

4. ***Rate-limited Background Load***

   Sometimes in order to limit the latency of a typical latency-sensitive job which is not throughput-bound, the background thread needs to be rate limited. This reduces the probability of interference of IO from an antagonist on the main workload. Due to this, the utilization of the disk drive is reduced by a large extent. Figure 6 plots latency of the foreground job while the throughput of the background job is increased. In this experiment, both jobs are requesting 64k-sized buffered reads sequentially to two different 2GB files on a 400GB drive for 300 seconds. The foreground job is doing reads at a fixed rate of 4096 kBps, while the throughput of the background job is varied from 100 kBps to the drive's maximum capacity. In absence of any *priority (No_Priority)*, both the mean and the maximum latency of the foreground job are affected when the throughput of the background job increases to around 4096kBps. But when the foreground job is run at a higher priority *(Priority)*, its latency is not affected with the increase in the rate of background job. Even when the background load is run at its maximum throughput of ~66 MBps, the foreground job is isolated. The drive is now able to service 16 times more background traffic, thus increasing its utilization.

Note: The next few experiments describe the performance of random reads in the presence of background traffic, as described in workload description. The first two experiments simulate a condition where these random reads are partly served from cache and there is sequential logging traffic competing with the foreground load. Various methods of doing sequential traffic are compared, with and without *priority*, to find out the best possible method which reduces the latency of random reads and which has good throughput.

Legend:

(a) no-load: Base case with no background logging traffic.

(b) direct-64k: Direct IO of block size 64k in background.

(c) buf: 64k-sized sequential buffered IO in background.

(d) buf-fadv: 64k-sized sequential buffered IO where the cache is dropped using `FADVISE_DONT_NEED` [17] after every 16MB.

(e) buf-prio: 64k-sized sequential buffered logging traffic running at lower priority.

(f) buf-prio-fadv: In addition to lower priority, sequential buffered antagonist has cache for the antagonist being dropped after every 16MB.

(g) direct-64k-prio: Direct IO of block size 64k at lower priority.

(h) buf-sync-16m: 64k sequential buffered background IO with sync every 16MB.

Type -	mean (msec)	max (msec)	std dev -	b/w MB/s
no-load	5.97	149	7.81	-
direct-64k	33.97	373	83.34	1.9
buf	111.78	528	160.05	0.6
buf-fadv	40.67	784	97.00	1.6
buf-prio	9.34	182	12.42	6.7
buf-prio-fadv	7.74	112	11.05	8.1
direct-64k-prio	7.48	116	10.32	8.3

Table 1: Random cached reads with background sequential reads

5. ***Random Cached Reads With Background Reads***

In this setup, 8GB of the 10GB file is read sequentially into memory and will form cache for the foreground thread doing 64k random reads. Table 1 shows the mean, max, and standard deviation of latency as well as the average bandwidth of the foreground job for various kinds of reads in the background. A random read in the presence of cached data performs worse when a sequential buffered read traffic (c) is running in background, since the logging traffic quickly destroys the cache and, in the absence of *priority*, competes with the foreground random traffic. Lower mean latency when using `fadvise` (d) confirms that the major portion of increase in latency for the buffered case comes from the cache being wiped out. Using lower priority either with buffered (e) or direct

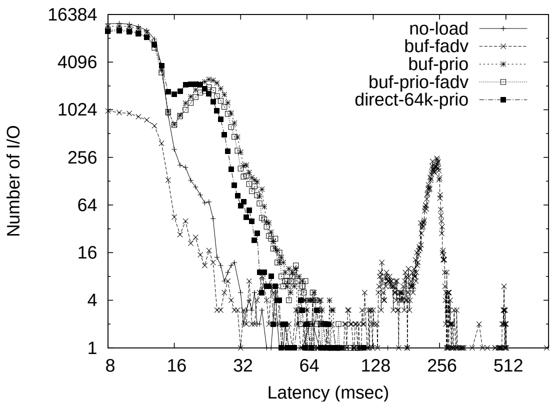

Figure 7: Foreground Cached Random reads – Background sequential reads

(g) reads in the background reduces both the average and tails of the foreground job. Reads which constantly drop cache and run at lower priority (f) further reduce the average and tails. Direct IO at lower priority (g) and buffered with cache drop (f) perform the best as they reduce the latency and at the same time increase bandwidth of the foreground job. Direct (g) is marginally better than (f).

Figure 7 shows the distribution of per-IO latency of the main thread. Even though Table 1 shows that buffered read running with lower priority alone (e) has a slightly longer tail as compared to when *priority* is used along with `fadvise` (f) or direct IO (g) for background threads, the graph clearly shows that running a lower priority read antagonist in all three cases (running alone (e), with `fadvise` (f), and with direct IO (g)) has almost similar distribution. Running `fadvise` alone (d) has a very long tail due to contention in the scheduler.

6. ***Random Cached Reads With Background Writes***

Similar to the last experiment, here also 8GB from a 10GB file are pre-cached into memory sequentially, followed by the main thread performing 64k random reads. In this case, the background load is sequential writes of various kinds, unlike in last experiment, where we were reading in the background. Here, direct IO (b) performs the worst due to contention in scheduler. Sequential (c) is a lot better, but dropping cache periodically; (d) improves the average latency further. Since the background writes can quickly wipe the cache, using *priority* alone (e) does not reduce the average as

Type	mean	max	std dev	b/w
-	(msec)	(msec)	-	MB/s
no-load	6.0	102	7.86	-
direct-64k	36.60	508	88.29	1.7
buf	12.89	891	30.83i	4.8
buf-fadv	7.49	236	15.68	8.3
buf-prio	12.15	258	15.10	5.1
buf-prio-fadv	8.17	133	12.06	7.6
buf-sync-16m	9.36	349	17.45	6.7
direct-64k-prio	6.27	99	8.60	9.9

Table 2: Random cached reads with background sequential writes

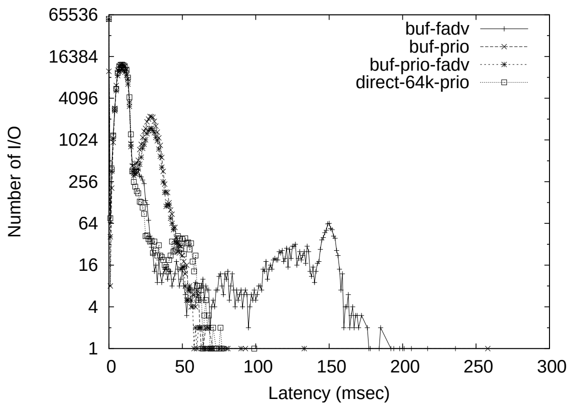

Figure 8: Foreground Cached Random reads – Background sequential writes

compared to buffered writes without *priority* (c). Similar to the results of the last experiment, lower priority direct writes (g) or buffered writes along with dropping cache (f) give the best performance both for latency (mean or max) and bandwidth. Using *priority* moves direct IO (g) from being the worst antagonist to least intrusive antagonist. Also, it reduces the tail for both direct (g) and buffered (f) logging traffic. Figure 8 has per-IO latency distribution for the main thread and shows the long tail of *buf-fadv* approach. Though Table 2 shows a higher value of max for *buf-prio*, Figure 8 clearly shows that for all cases using *priority* (e, f, g), the tails end around 100msec. Doing sync alone (h) does not perform as well as `fadvise` (d) due to cache effects.

7. ***Random Reads With Background Reads***

Unlike the previous two experiments, there is no

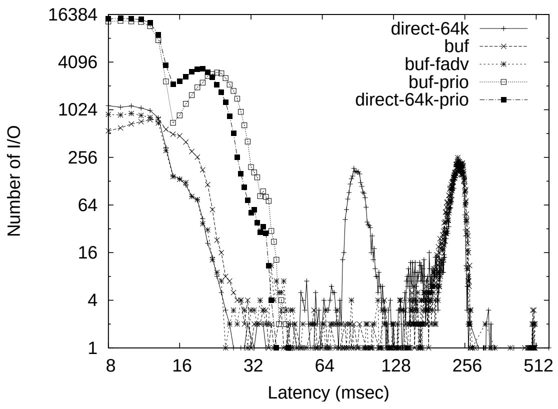

Figure 9: Foreground Random reads – Background sequential reads

sequential caching read in the next two experiments. In Table 3 we measure the latency and throughput of a thread doing 64k random reads with different kinds of antagonists reading sequentially. Unlike in the case of caching reads, `fadvise` does not help here, both mean and maximum latency when using `fadvise` (d) is almost similar to buffered reads (c). Using *priority* (e) reduces the average and maximum latency by eliminating contention in the scheduler queues. It helps in increasing bandwidth and reducing latency for both direct (g) as well as buffered IO (e). Direct IO seems to be slightly better, but the difference between it and buffered IO is within experimental noise. Figure 9 shows the per-IO latency distribution and in all cases where *priority* is used (e, f, g), tails are shortened by a large amount.

8. ***Random reads with background writes***

Type	mean	max	std dev	b/w
-	(msec)	(msec)	-	KB/s
no-load	8.91	32	9.34	6959
direct-64k	93.46	579	138.14	714
buf	115.29	506	161.29	577
buf-fadv	113.87	507	161.08	583
buf-prio	11.88	47	13.59	5294
buf-prio-fadv	11.47	63	13.05	5475
direct-64k-prio	11.13	46	12.34	5633

Table 3: Random reads with background sequential reads

Type	mean	max	std dev	b/w
-	(msec)	(msec)	-	KB/s
no-load	9.16	105	9.66	6782
direct-64k	100.94	512	145.47	654
buf	12.93	698	30.01	4939
buf-fadv	11.39	408	19.36	5516
buf-prio	13.09	155	15.66	4825
buf-prio-fadv	12.53	325	15.03	5033
buf-sync-16m	11.47	218	19.14	5482
direct-64k-prio	9.66	96	10.67	6450

Table 4: Random reads with background sequential writes

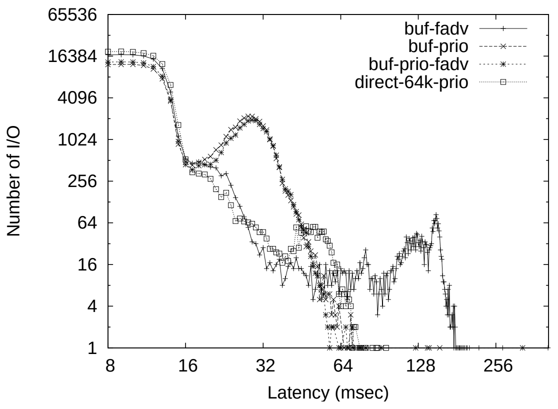

Figure 10: Foreground Random reads – Background sequential writes

Table 4 has the results of running a 64k random read thread along with various kinds of sequential write threads running in the background doing 64k-sized IO. This is similar to experiment 6, but with no caching reads. Unlike experiment 6, buffered writes (c) in the background are better than direct IO (b), since in this case wiping the cache does not have an impact on the foreground random read thread. Using direct IO (b) without priority causes heavy contention in the scheduler, unlike the normal write requests. In fact, using priority (e) or `fadvise` (d) has negligible impact on the mean latency, though the latency tail is reduced to a large extent by using *priority*. The large maximum latency in Table 4 when using both `fadvise` and priority (f) is due to an outlier. Looking at Figure 10 it is clear that using *priority* eliminates long tails for all kinds of logging workloads (e, f, g). And like experiment 6, direct IO has changed from being the worst antagonist to the least intrusive one.

5 Conclusion and Future Work

Using *priorities* in a *non-work-conserving* scheduler, we have been able to isolate latency of both buffered and direct high-priority synchronous requests. Increasing the number of background jobs has small/ negligible impact on latency. Even though we tagged asynchronous requests, multiple writers still perform non-deterministically. For our example workload's (1) cache-sensitive reads, interference from background reads can be effectively isolated using *priority*. In the case of background writes, using *priority* along with cache drop hints or direct IO reduces the average. For random reads (2), *priority* is able to handle reads/writes as antagonists. Using *priority* we can effectively eliminate tails in all cases.

In our solution we either bypassed cache (direct IO) or used explicit hints to drop cache (`fadvise`) for background writes, but in a generic solution apart from feedback controlled IO throttling, we need prioritized submission in background writes like `pdflush`. Though we used `page struct` as a prototype to tag asynchronous *priorities*, in context of `cgroups` we could use `ioprio` of a control group associated with the page as proposed by Fernando [8]. Another non-container solution would be to use `io_context` to hold this information.

There is a push to create an IO subsystem controller for dividing available bandwidth among various cgroups. The Linux kernel mailing list had a proposal to limit the bandwidth [20] based on values configured in a control group filesystem. [28] [12] [25][9] are proportional bandwidth-scheduling solutions, while [12] is an IO scheduler solution, [25] is the solution in the device-mapper driver. [26][24] are attempts in the `cfq` scheduler to add fairness control per group rather than per process. While we agree that dividing bandwidth proportionally is needed for I/O controller, we would also like to be able to attach *priority* to a process or control group. This is a more feasible solution than adding latency and bandwidth requirements to a control group. In this solution, a group having higher priority would be given preference within its quota of bandwidth.

Proposed interface for such a scheduling scheme

1. *Using cgroup interface*: Add *blockio.bandwidth* and *blockio.priority* files per cgroup.

   /dev/cgroup/group1/blockio.bandwidth 20

   /dev/cgroup/group1/blockio.priority 0

   /dev/cgroup/group1/blockio.prios_allowed BE(2–4)

   In this scheme, all processes submitting IO within `group1` get 20% of available bandwidth, while being the highest priority (0) they are served before processes from any other group. Moreover, processes in `group1` are allowed to use only *best-effort* levels 2–4.

2. *Using ionice approach*: When not using cgroups, for per-process bandwidth/latency control, `ioprio_set()` / `ioprio_get()` can be used to specify bandwidth as well. This is overloading of the current definition used by *CFQ*, but we could add another class not conflicting with ones defined now.

6 Acknowledgements

Thanks to Mike Waychison, Grant Grundler, Shishir Verma, Paul Menagae, and Al Borchers for reviewing this document or providing technical guidance. Special thanks to Mike Waychison for code reviews and discussions.

References

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Workshop on Internet Server Performance*, pages 91–102, 1998.

[2] Jens Axboe. Fio - flexible io tester. `http://freshmeat.net/projects/fio/`.

[3] Jens Axboe. [patch][cft] time sliced cfq ver18. `http://lkml.org/lkml/2004/12/21/67`.

[4] Jens Axboe. Time sliced cfq io scheduler. `http://lwn.net/Articles/113869/`.

[5] Jens Axboe. Linux block io – present and future. In *Ottawa Linux Symposium*, pages 51–61, 2004.

[6] J. Bruno, E. Gabber, B. Ozden, and A. Silberchatz. The eclipse operating system: Providing quality of service via reservation domain. In *USENIX Annual Technical Conference*, pages 235–246. USENIX, 1998.

[7] John Bruno, Jose Brustoloni, Eran Gabber, Banu Ozden, and Abraham Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, page 400, 1999.

[8] Fernando Luis Vazquez Cao and Hiroaki Nakano. Cfq vs containers. `http://iou.parisc-linux.org/lsf2008/IO-CFQ_vs_Containers-Fernando_Luis_V%e1zquez_Cao.pdf`.

[9] Fabio Checconi. Bfq i/o scheduler. `http://lkml.org/lkml/2008/4/1/234`.

[10] Giel de Nijs, Benno van den Brink, and Werner Almesberger. Active block i/o scheduling system. In *Ottawa Linux Symposium*, pages 109–126, 2004.

[11] Ajay Gulati, Arif Merchant, and Peter J. Varman. pclock: an arrival curve based approach for qos guarantees in shared storage systems. In *SIGMETRICS*, pages 3–24, 2007.

[12] Naveen Gupta. [rfc] proportional bandwidth scheduling using anticipatory i/o scheduler. `http://lkml.org/lkml/2008/1/30/10`.

[13] Christoffer Hall-Frederiksen and Philippe Bonnet. Using prioritized i/o to improve storage bandwidth in mysql. VLDB 2005.

[14] Sitaram Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *SOSP*, 2001.

[15] W. Jin, J. Chase, and J. Kaur. Interposed proportional sharing for a storage service utility. In *ACM Sigmetrics -Performance*, 2004.

[16] C. Lumb, A. Merchant, and G. Alvarez. Facade: Virtual storage devices with performance guarantees. In *Conference on File and Storage Technology*, pages 131–144, 2003.

[17] Andrew Morton. Usermode pagecache control: fadvise(). `http://lkml.org/lkml/2007/3/3/110`.

[18] Nick Piggin. Anticipatory i/o scheduler. `http://lxr.linux.no/linux/Documentation/block/as-iosched.txt`.

[19] Lars Reuther and Martin Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset. In *24th IEEE International Real-Time Systems Symposium*, page 374, 2003.

[20] Andrea Righi. cgroup: limit block i/o bandwidth. `http://lkml.org/lkml/2008/1/18/166`.

[21] S. Seelam, J. Babu, and P. Teller. Automatic i/o scheduler selection for latency and bandwidth optimization. In *Workshop on Operating System Interference in High Performance Applications*, 2005.

[22] S. Seelam and P. Teller. Virtual i/o scheduler: a scheduler of schedulers for performance virtualization. In *3rd international conference on Virtual execution environments*, pages 105–115, 2007.

[23] Seetharami R. Seelam, Andre Kerstens, and Patricia J. Teller. Throttling i/o streams to accelerate file-io performance. In *HPCC*, pages 718–731, 2007.

[24] Vasily Tarasov. cgroups: block: cfq: I/o bandwidth controlling subsystem for cgroups based on cfq. `http://lkml.org/lkml/2008/3/21/519`.

[25] Ryo Tsuruta. dm-band: The i/o bandwidth controller. `http://lkml.org/lkml/2008/1/23/106`.

[26] Satoshi UCHIDA. Yet another i/o bandwidth controlling subsystem for cgroups based on cfq. `http://lkml.org/lkml/2008/4/3/45`.

[27] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R. Ganger. Argon: Performance insulation for shared storage servers. In *5th USENIX Conference on File and Storage Technologies*, pages 61–76, 2007.

[28] C. Waldspurger and W. Weihl. Stride scheduling: Deterministic proportional-share resource management, 1995.