Throttling On-Disk Schedulers to Meet Soft-Real-Time Requirements *

Mark J. Stanovich, Theodore P. Baker, An-I Andy Wang
Department of Computer Science
Florida State University
Tallahassee, FL 32306-4530
e-mail: [stanovic, baker, awang]@cs.fsu.edu

Abstract

To achieve better throughput, many hard drive manufacturers use internal queues and scheduling to take advantage of vendor-specific characteristics and knowledge. While this trend seems promising, the control of service time from the view of the operating system is weakened, posing challenges for scheduling requests that have real-time requirements. Also, the diversity of disk drives makes extracting detailed timing characteristics and its generalization for all hard drives very difficult. This paper demonstrates three techniques we developed under Linux to bound real-time request response times for disks with internal queues and schedulers. One is to use the disk's built-in starvation prevention scheme. The second is to prevent requests from being sent to the disk when real-time requests are waiting to be served. The third, limits the length of the on-disk queue in addition to the second technique. Our results show the ability to guarantee a wide range of desired response times while still allowing the disk to perform scheduling optimizations.

1 Introduction

The request service time of a mechanical disk drive is many orders of magnitude slower when compared to the majority of other electronic components of a computer. To minimize the mechanical movements of the disk and subsequently improve its performance, one common optimization is to reorder requests. For many years, the I/O scheduler component of an operating system has been responsible for providing request reordering to achieve high throughput and a low average response time while avoiding starved requests.

Typical operating systems have a general notion of the disk's hardware characteristics and interact with disks through a rigid interface; however, the detailed data layout and capabilities of the disk are generally hidden. For instance, it may be believed that the disk's logical block addresses (LBA) start from the outside perimeter of the disk and progress inwards, but [12] has observed that LBA 0 on some Maxtor disk drives actually starts on track 31. Another example is the common perception that issuing requests with consecutive addresses will give the best performance. Again, some disks support zero-latency access, which permits the tail end of a request to be accessed before the beginning, so that the disk head can start transferring data as soon as a part of the request is under the disk head, not necessarily at the beginning. This out-of-order data access scheme reduces the rotational delay to wait for the beginning of the data request to be positioned under the disk head before the data transfer starts [16].

Given that the operating system has limited knowledge of the layout, capabilities, timing characteristics and real-time state of individual disk drives, disk manufacturers provide tailored optimizations such as built-in schedulers to better exploit vendor-specific knowledge. To schedule requests on-disk, a drive needs to provide an internal queue that can take multiple requests from an operating system and further reorder requests specific to a particular disk drive. Figure 1 illustrates the

*Based upon work supported in part by the National Science Foundation under Grant No. 0509131, and a DURIP equipment grant from the Army Research Office.

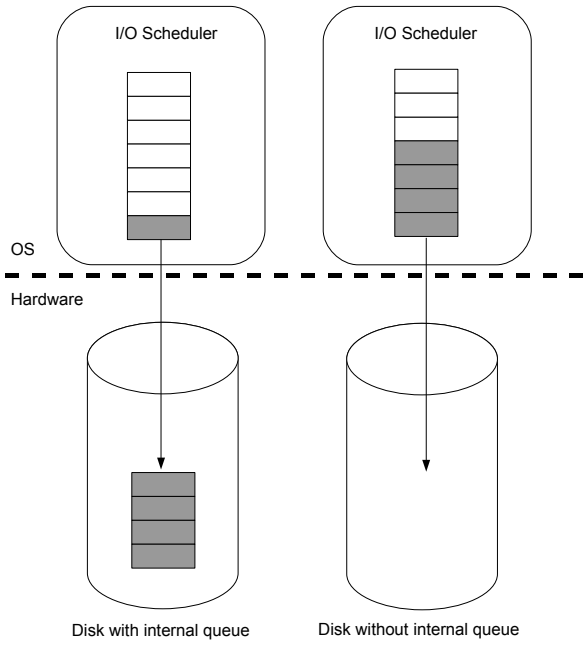

Figure 1. Alternative disk-request queuing schemes.

key difference between disks with and those without an internal queue/scheduler. Instead of maintaining all requests in the I/O scheduler framework, disks with a built-in queue allow multiple requests to be issued to the disk without waiting for the completion of a previous request. This permits multiple requests to be pending at the operating system level as well as the hardware level. Both locations allow reordering of requests; however, once requests have been sent to the disk drive, control over their service order shifts from the operating system to the built-in disk scheduler.

Although on-disk schedulers have shown promise, they introduce concerns for real-time systems. Since the disk can change the ordering of requests, the individual request service times can be difficult to control and predict from the viewpoint of an operating system. Instead of having to determine the completion time one request at a time, an operating system needs to predict the order the disk will serve multiple requests in order to provide response time guarantees. Further, worst-case service times of requests sent to the disk can be increased to many times that of a disk that does not contain an internal queue. To use these disks to serve real-time requests, we need to address all these concerns.

2 Motivation

Certain characteristics of disk drives make it difficult to predict I/O response times accurately. One is the variability of service times due to the state of the disk caused by a prior request. With disks that allow one outstanding request at a time, a new request to disk from a device driver must wait until the completion of the previous request. Only then can a new request be issued from a device driver to disk. Next, based on the location of the previous request, the disk must reposition the head to a new location. Given these factors, the variability of timings can be on the order of tens of milliseconds.

Many disks now also contain an extra state parameter in the form of an internal queue. This is available on the majority of SCSI and most new SATA drives. Command queuing is the protocol used to send multiple requests to the hard drive, with three common policy variants: simple, ordered, and head of queue [1]. When a new request is sent to the disk, one of the policies must be specified with a tag. The simple tag indicates that the request may be reordered with other requests marked as simple. The ordered tag specifies that all older requests must be completed before the ordered request begins its operation. The ordered requests will then be served followed by any remaining ordered or simple tagged commands. Lastly, the head-of-the-queue tag specifies that the request should be the next command to be served after the current command (if it exists).

With an internal queue, the variability in request service time is significantly larger. Once a request is released to the disk for service, the time-till-completion will depend on the service order of the queued requests established by the disk's internal scheduler and the given insertion policy. Now, instead of having to wait tens of milliseconds for a particular request to return, the maximum service time can be increased to several seconds.

2.1 Observed vs. Theoretical Bounds

To demonstrate and quantify problems of real-time disk I/Os resulting from drives with internal queues/schedulers, we conducted some simple exper-

Hardware/software	Configurations
Processor	Pentium D 830, 3GHz, 2x16KB L1 cache, 2x1MB L2 cache
Memory	1.5GB
Hard disk controller	Adaptec 4805SAS
Hard disks	Maxtor ATLAS 10K V, 73GB, 10,000 RPM, 8MB on-disk cache, SAS (3Gbps) [11]
	Fujitsu MAX3036RC, 36.7GB, 15,000 RPM, 16MB on-disk cache, SAS (3Gbps) [9]
	IBM 40K1044, 146.8GB, 15,000 RPM, 8MB on-disk cache, SAS (3Gbps) [7]
Operating system	Linux 2.6.21-RT PREEMPT

Table 1. Hardware and software experimental specifications.

iments. These tests were run on the RT Preempt version of Linux [2], which is standard Linux patched to provide better support for real-time applications. The hardware and software details are summarized in Table 1.

To estimate service times for a particular request by a real-time application, one could make simple extrapolations based on the datasheets for a particular drive. Considering disk reads, a naive worst-case bound could be the sum of the maximum seek time, rotational latency, and data access time. According to the datasheet for the Fujitsu drive [9] the maximum seek time and rotational latency are 9 and 4 milliseconds respectively. Assuming that the disk head can read as fast as the data rotates underneath the head, the data transfer time would then be the time spent rotating the disk while reading the data, which is a function of the request size. For our experiments, we chose a 256KB request size. Since modern drives store more information on outer tracks than inner tracks, this chosen access granularity corresponds to half to sometimes more than one track on various disks. This means that potentially, a request could take another rotation to access the data, which adds an extra 4 msec, resulting in a worst-case bound of 17 msec. Clearly, this estimation is crude and overlooks factors such as settling time, thermal recalibration, read errors, bad sectors, etc. However, the aim is to develop a back-of-the-envelope intuition on the range of expected service times.

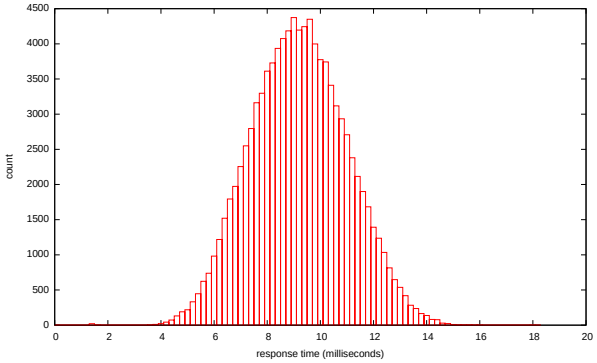

Figure 2. Observed disk completion times with no interference.

To validate these coarse estimated service times empirically, we created a task that makes 256-KB disk requests at uniformly distributed random locations. Each request's completion time was calculated and plotted as shown in Figure 2. The first observation is that almost all requests were completed within the predicted 17 msec time frame. However, a few requests exceeded the maximum expected completion time, the latest being 19 msec. These outliers could be attributable to any of the above mentioned causes that were not included in our coarse estimation method. Using the observed maximum completion time of 19 msec, it appears that using disks for soft-real-time applications is quite plausible.

2.2 Handling Background Requests

The above bound, however, does not consider the common environment with mixed workloads, where real-time requests and best-effort requests coexist. This mixture increases the worst-case service completion time of the disk. On a disk that can accept one request at a time, this worst-case completion time for a request could be estimated at twice the maximum completion time for one request, that is: one time period for a request being served, which cannot be preempted; another time period to serve the request. However, disks with internal queues show a very different picture.

Internal queues allow for multiple requests to be sent to the disk from the device driver without having to wait for previous requests to be completed. Given several requests, the disk's built-in scheduler is then able to create a service order to maximize the efficiency of the disk. This, however, poses a problem with requests that possess timing constraints, since meeting timing constraints can be at odds with servicing the given requests efficiently.

Take a scenario with both real-time and best-effort requests. For real-time requests, the I/O scheduler at the device driver level should also support real-time capabilities. That is, real-time requests should be sent to the disk before any best-effort requests. Without real-time capabilities, a backlog of best-effort requests could easily accumulate, resulting in high I/O latencies. While Linux does have an interface to set the I/O priorities, only the complete fairness queueing (CFQ) scheduler applies any meaning to the value of these priorities. Using the CFQ scheduler is an option; however, it incurs additional complexity that further hinders the understanding of disk drive service times.

2.3 Prioritized Real-Time I/Os are not Enough

To investigate the latencies associated with using disk drives, we implemented a basic real-time I/O (RTIO) scheduler in Linux. This scheduler respects the priorities set by the individual applications. RTIO performs no request merging. Also, the requests within individual priority levels are issued in a first-come-first-served fashion. All disk-location-based sorting and merging are handled internally by the disk's built-in scheduler.

With RTIO, we designed an experiment to measure real-time request latencies where two processes generated 256-KB read requests to random locations on disk. One is a real-time task that repetitively performs a read and waits for its completion. The second process is a best-effort process that generates up to 450 asynchronous read requests at a time. The idea is to generate significant interference for the real-time task. The results for the completion times of the real-time requests are graphed in Figure 3, using a Fujitsu drive. The reordering of requests by the disk can cause unexpectedly long response times. The largest observed latency was around 1.9 seconds. The throughput, however, was respectable, at 39.9 MB/sec[1]. This tradeoff seems to be favored by the on-disk internal scheduler. Although ideally one would like high throughput and low average response time, these two goals are often in conflict with each other.

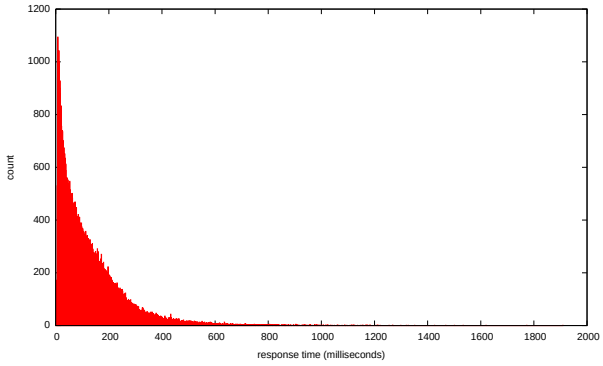

Figure 3. Completion times for real-time requests in the presence of background activity using the RTIO scheduler.

Disks with an internal scheduler/queue can significantly increase the variance of I/O completion times and reduce their predictability than without. However, the internal queue does provide some benefits. Not only does it provide good throughput, but it also allows the disk to remain busy while waiting for requests to arrive from the device driver queue. Particularly, in the case of real-time systems, the code for the hard disk data path might have a lower priority than other tasks on the system, causing delays in sending requests from the device driver to the disk, to keep the disk busy. Without an internal queue, a disk will become idle, impacting on both throughput and response times of disk requests. The severity depends on the blocking time of the data path. Even with an internal queue, the problem of reducing and guaranteeing disk response time remains. Without addressing these issues it is unlikely

[1] The maximum observed throughput of the disk on other experiments in which we fully loaded the disk with sequential read requests ranged from 94.7 to 73.0 MB/sec, depending on the cylinder. Clearly, that level of throughput is not possible for random read requests. A rough upper bound on random-access throughput can be estimated by taking the request size and dividing it by average transfer time, seek time for 20 requests, and rotational delay. For our experiment this is $256KB/(3msec + .55msec + 2msec)$ giving a throughput of 46.1 MB/sec. This is not far from the 40 MB/sec achieved in our experiments.

anyone would choose to use a disk in the critical path of real-time applications.

3 Bounding Completion Times

This section describes the various ways in which we explored bounding the completion times of real-time requests that were sent to the hard disk. These include using the built-in starvation prevention algorithm on the disk, limiting the maximum number of outstanding requests on the disk, and preventing requests being sent from the device driver to the disk when completion time guarantees are in jeopardy of being violated.

3.1 Using the Disk's Built-in Starvation Prevention Schemes

Figure 3 shows that certain requests can take a long time to complete. There is, however, a maximum observed completion time at around two seconds. This may reflect that the disk is aware a request is being starved, and it forces the request to be served even though it is not the most efficient request to serve next. To test this hypothesis, we created a test scenario that would starve one request for a potentially unbounded period of time. That is, one requested disk address would be significantly far away from the others, and servicing the outlier request would cause performance degradation. Better performance would result if the outlier request were never served. The point at which the outlier request returns would be the maximum time a request could be queued on a disk without being served.

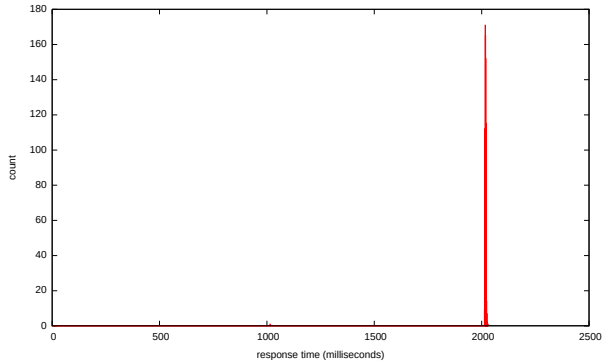

Figure 4. Disk's observed starvation prevention.

To create this scenario, we designed an experiment where best-effort requests were randomly issued to only the lower 20 percent of the disk's LBAs. At the same time, one real-time request would be issued to the disk's upper 20 percent address space. A maximum of twenty best-effort requests, and one real-time request, were permitted to be queued on the disk at one time. The idea was that the disk would prefer to serve the best-effort requests, since the access locations are closer to one another and would yield the best performance. Figure 4 shows the results of this experiment on a Fujitsu drive. The spike, at just over 2 seconds, appears to be the maximum request time able to be generated by the disk. Given this information, real-time applications that require a completion time greater than 2.03 seconds need nothing special to be done. The on-disk scheduling will provide 40 MB/sec, while preventing starved requests. Intriguingly, the spike is cleanly defined, and suggests that the disk has a strong notion and control of completion times, rather than counting requests before forcing a starved request to be served.

Should a real-time application require lower completion time than the disk-provided guaranteed completion time, additional mechanisms are needed.

3.2 "Draining" the On-Disk Queue

Reordering of the requests currently on the disk's queue is not the only source of problems that cause extended completion times in Figure 3. As on-disk queue slots becomes available, newly arrived requests can potentially be served before the previously sent requests, which explains why the completion times are greater than that of just servicing the number of possible requests permitted to be queued on a disk.

To determine the extent of starvation due to continuous arrivals of new requests, we first flooded the on-disk queue with 20 best-effort requests, then measured the time it takes for a real-time request to complete without sending further requests to the disk. As anticipated, Figure 5 shows that the completion times are significantly shorter. Contrary to our intuition, more real-time requests are served with a shorter completion time, while it seemed the real-time request should have had an equal chance of being chosen to be the next request to be served, among all the requests. With an in-

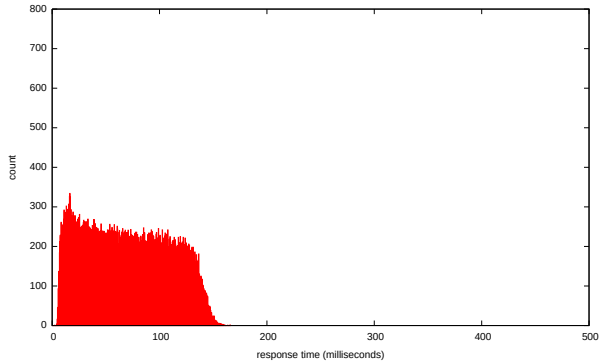

Figure 5. Effect of draining requests on the Fujitsu hard disk.

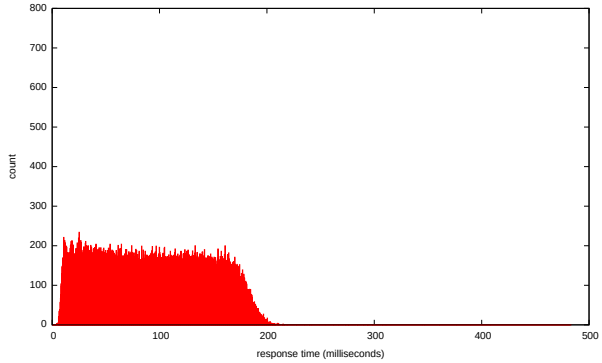

Figure 6. Effect of draining requests on the Maxtor hard disk.

house simulated disk, we realized that with command queuing, the completion time reflects both the probability of the real-time request being chosen as the nth request to be served, as well as the probability of various requests being coalesced in ways to be served with

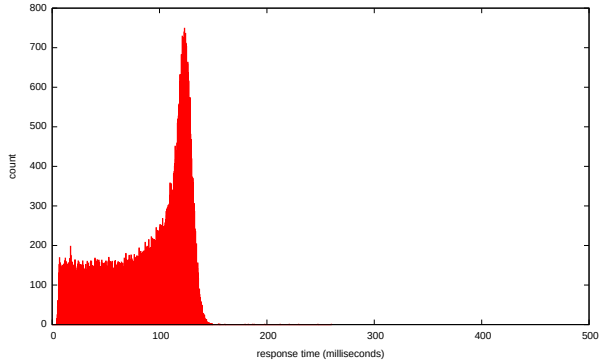

Figure 7. Effect of draining requests on the IBM hard disk.

fewer rotations. In this case, the probability of serving a real-time request as the last request while taking all twenty rotations is rather unlikely.

Applying the "draining" mechanism used for this experiment, we have a new means to bound completion times. Draining gives the disk fewer and more bounded choices to make when deciding the next request to serve. As each best-effort request returns, the likelihood for the disk to serve the real-time request increases. In the worst-case, the real-time request will be served last.

Using the worst-case drain time for a permitted on-disk queue length, we can bound the completion time for a real-time request. Note that draining may not be required, if the deadline is not jeopardized. Also, once the real-time request is served, the device driver can continue to send new requests to disk without completely draining the queue.

While draining can prevent completion time constraints from being violated, it relies on knowing the worst-case drain time for a given number of outstanding requests on disk. Predicting this time can be difficult. One approach is to deduce the maximum possible seek and rotation latencies, based on possible service orderings for a given number of requests. However, the built-in disk scheduler comes preloaded in the firmware, with undisclosed scheduling algorithms. Also, our observations show that on-disk scheduling exhibits a mixture of heuristics to prevent starvations. To illustrate, Figures 5, 6, and 7 used the same draining experimental framework on disks from different vendors. From the diversity of completion-time distributions, the difficulty of determining the scheduling algorithm is evident. Further, even if one drive's scheduling algorithm is discovered and modeled, this does not generalize well with the diversity and rapid evolution of hard drives.

Given these issues, simple and general analytical prediction of the drain time may not be realistic. However, the maximum drain time can be determined empirically with a relatively high confidence level. To obtain the drain time for a given number of requests x, an experiment can be performed by sending x random requests to the disk with a uniform distribution across the entire disk. The time for all requests to return is then logged. The graph of the experiment for the drain time of 20 requests on the Fujitsu drive is shown

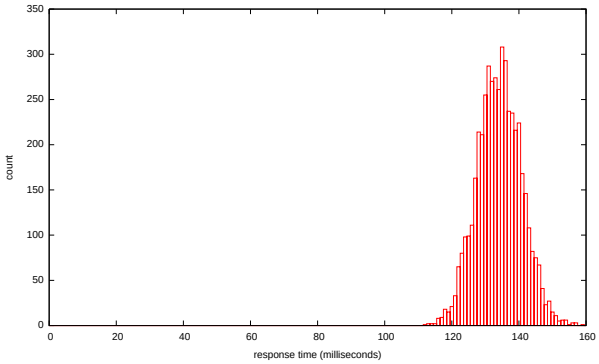

Figure 8. Empirically determining the drain time for 20 outstanding disk requests on the Fujitsu drive.

in Figure 8, which will allow us to bound the completion time for a real-time request with the presence of 19 outstanding best-effort requests.

3.3 Experimental Verification

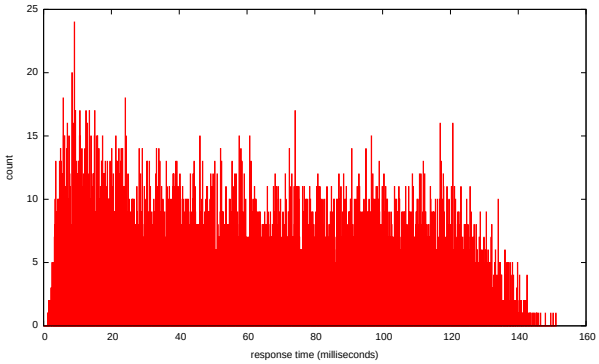

Figure 9. Draining the queue to preserve completion times of 160 msec.

To implement the proposed draining policy, our RTIO scheduler was modified to stop requests being issued from the device driver to the disk once a real-time request has been issued from a device driver. If no real-time requests are present, the scheduler limits the maximum number of on-disk best-effort requests to 19. For the experiment, we created two processes. One process was a periodic real-time task that read 256 KB from a random disk location every 160 msec, which was also the deadline. The deadline was based on maximum drain time in Figure 8. The other process is a best-effort task that continuously issues 256-KB asynchronous read requests with a maximum of 450 outstanding requests. Figure 9 shows that no completion times exceeded 160 msec, and no deadlines were missed. The throughput remained at 40 MB/sec, suggesting that draining the entire queue occurred rather infrequently.

3.4 Limiting the Effective Queue Depth

While draining helps meet one specific completion time constraint (e.g., 160 msec), configuring draining to meet arbitrary completion times (e.g., shorter deadlines) requires additional mechanisms. One possibility is to further limit the number of outstanding requests on disk. This effectively limits the queue depth of the disk, thereby reducing maximum drain times. By determining and tabulating the drain time for various queue lengths (Figure 8), we can then meet arbitrary completion time constraints (of course subject to the timing limitations of physical disks).

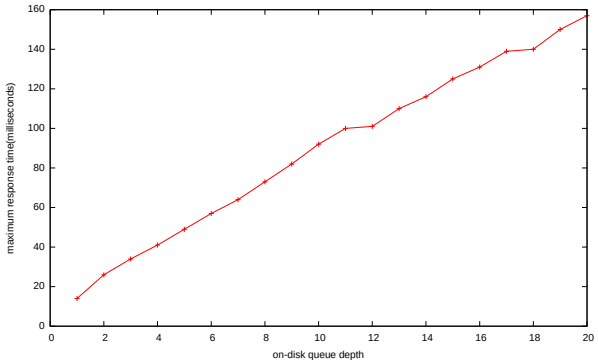

Figure 10. Maximum observed drain times for on-disk queue depths.

For instance, to meet the response time constraint of 75 msec, Figure 10 shows that using a queue depth of less than or equal to 7 would suffice. We would like to use the largest possible queue depth while still maintaining the desired completion time for real-time I/O requests. A larger queue length allows the disk to have more requests to choose from and as a result make better scheduling decisions. Therefore, in this case, we would choose a queue depth of 7. The best-effort tasks must be limited to a queue depth of 6 and one slot will be reserved for the real-time request.

To verify our tabulated queue length, a similar ex-

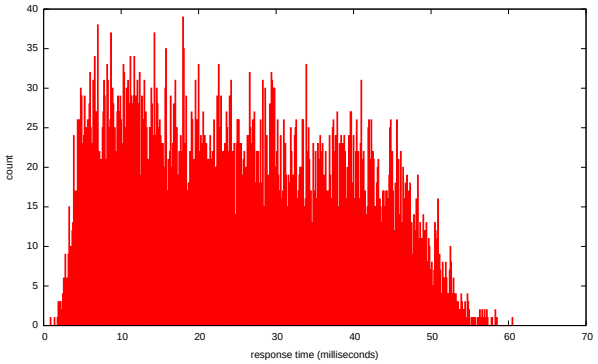

Figure 11. Limiting and draining the queue to preserve completion times of 75 msec.

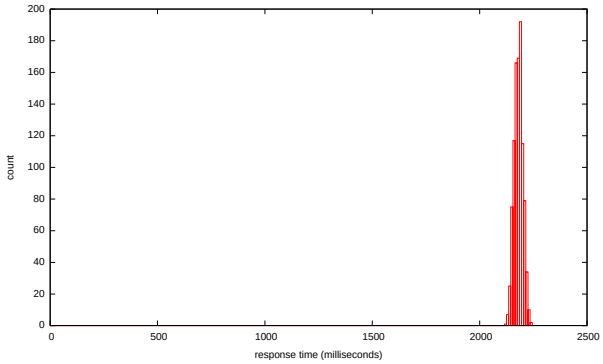

Figure 12. Completion times for real-time requests using the Linux CFQ I/O scheduler.

periment was performed as before, with 75 msec as the period and deadline. While Figure 11 shows that all deadlines are met as expected, we noticed that the throughput (not shown) for the best-effort requests dropped to 34 MB/sec. Interestingly, a 70% drop in queue length translates into only a 15% drop in bandwidth, demonstrating the effectiveness of on-disk queuing even with a relatively short queue.

4 Comparisons

To show the benefits of our approach to bound I/O completion times, we compared RTIO with the CFQ I/O scheduler, the default I/O scheduler for Linux. CFQ was chosen since it is the only standard Linux scheduler that uses I/O priorities when making scheduling decisions. Without this support, it is easy to see situations where a backlog of best-effort requests at the device driver level may prevent a real-time request from reaching disks until they are served, in addition to the requests queued on disk. Given that the combined queue depth can be quite large, potentially a real-time request may be forced to wait for hundreds of other requests to complete.

To get an idea of the worst-case completion times of real-time requests sent to the disk using the CFQ scheduler, we repeated similar experiments as in Figure 3. These experiments provided a continuous backlog of 450 asynchronous best-effort requests to random locations on disk, in addition to one real-time request sent periodically. The request size was again limited to 256KB. The only difference was to use CFQ rather than RTIO. Figure 12 shows that real-time completion times can exceed 2 seconds. Without the limit imposed by the disk's starvation control algorithm, the real-time response might have been even worse. The poor real-time performance is because CFQ continuously sends best-effort requests to the disk, even though there may be real-time requests waiting to be served on the disk. Since the disk does not discern real-time and best-effort requests once they are in the on-disk queue, many best-effort requests can be served before serving the real-time request. At first it may seem peculiar that the majority of requests are over 2 seconds, considering the requests are randomly distributed across the entire disk. This problem occurs because CFQ sorts the random collection of best-effort requests, resulting in requests being sent to the disk which are likely to be closer to each other than to the real-time request. This results in the disk servicing the best-effort requests prior to the real-time request to obtain the most efficient order.

Surprisingly, the longer completion times do not translate into better throughput. We observed 27.3 MB/sec under CFQ, which is 20% to 30% lower than previous experiments. This may be attributable to poor interactions between the CFQ scheduler and the on-disk scheduler.

5 Related Work

Many researchers have investigated scheduling real-time and best-effort hard disk requests. Some examples include Shenoy and Vin [15], Bosch, Mullender and

Jansen [5], and Bosch and Mullender [4]. However, the majority of such studies do not consider the reordering effect of the internal disk scheduler. Also, many – for example, see Reddy and Wyllie [13] and Cheng and Gillies [6] – require detailed knowledge of the disk's internal state, as well as its layout and timing characteristics. Such information is becoming more and more difficult to obtain with the rapid growth in complexity and evolution of disk drives.

Reuther and Pohlack [14] and Lumb, Schindler and Ganger [10]. They have been able to extract disk characteristics and perform fine-grained external scheduling. They show that they can out-perform the on-disk scheduler in some cases. However, determining such timing information from disks can be very challenging and time-consuming, and can be expected to become more so as disk drives become more sophisticated.

Fine-grained CPU-based disk scheduling algorithms require that the disk device driver keep accurate track of the disk's state. In a real-time system, device drivers compete for CPU time with hard-real-time application tasks [18, 3]. Therefore, it may be necessary to schedule the processing of disk I/O requests by the device driver at a lower priority than some other tasks on the system, Interference by such tasks may prevent a CPU-based disk scheduling algorithm from keeping accurate track of the disk's state in real time, even if the layout and timing characteristics of the disk are known. Also, if the on-disk queue is not used, there is risk of idling the disk while it waits for the next request from the CPU. This may affect the utilization of the disk with a severity proportional to the amount of time that the device driver is blocked from executing.

Internal disk scheduling algorithms do not have these problems, since they are executed by a dedicated processor inside the disk, with immediate access to the disk's internal state and timing characteristics. Even if the device driver is run at maximum priority, an off-disk scheduler will have less complete and less timely information, and less precise control, due to the limited information provided by the disk I/O interface, and contention and transmission delays through the intervening layers of bus and controller.

The need to consider the internal scheduler of disks has been discussed in [8], which uses a round-based scheduler to issue requests to the disk. This allows real-time requests to be sent to the disk at the beginning of each round. The SCSI ordered tag is then used to force an ordering on the real-time requests. This approach prevents interference of requests sent after the real-time requests. However, it forces all real-time requests to be present at the beginning of the round. If the arrival of the real-time requests just misses the beginning of the round, the worst-case response times can be just under two rounds. Further, using the ordered tag may impose a first-come-first-served policy on the disk even when missed deadlines are not in jeopardy, which reduces the flexibility of the disk to make scheduling decisions and decreases the performance.

Another thread of research on real-time disk scheduling is represented by Wu and Brandt [17]. Noting the increasing intelligence of disk drives, they have taken a feedback approach to scheduling disk requests. When a real-time application misses it's deadline, the rate of issuing the best-effort requests is reduced. While their work provides a way to dynamically manage the rate of missed deadlines, they do not provide precise *a priori* response time guarantees.

6 Conclusion

In this paper, we discussed how to use the hard disk's internal queue and scheduler without jeopardizing response time constraints for real-time requests. While this goal may also be achieved by the operating system, allowing the disk to make request scheduling decisions alleviates the burden off the device-driver-level I/O scheduling and permits some optimizations that cannot be achieved in the operating system. Our explored approaches allow a disk to perform the work with its intimate knowledge of low-level hardware and physical constraints. Therefore, the disk can have a more informed access to its near-future request information while reordering requests to realize efficient use of the disk's resource. Further, the disk can achieve a higher level of concurrency with CPU processing, servicing on-disk requests without immediate attention from the operating system. This allows high-priority real-time processes to use the CPU with little impact on disk performance.

References

[1] Scsi architecture model - 3 (SAM-3). http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf, 2004.

[2] RT PREEMPT. http://www.kernel.org/pub/linux/kernel/projects/rt/, 2007. [Online; accessed 04-October-2007].

[3] T. P. Baker, A.-I. A. Wang, and M. J. Stanovich. Fitting linux device drivers into an analyzable scheduling framework. In *Proceedings of the 3rd Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, July 2007.

[4] P. Bosch and S. J. Mullender. Real-time disk scheduling in a mixed-media file system. In *RTAS '00: Proceedings of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, page 23, Washington, DC, USA, 2000. IEEE Computer Society.

[5] P. Bosch, S. J. Mullender, and P. G. Jansen. Clockwise: A mixed-media file system. In *ICMCS '99: Proceedings of the IEEE International Conference on Multimedia Computing and Systems Volume II-Volume 2*, page 277, Washington, DC, USA, 1999. IEEE Computer Society.

[6] R. M. K. Cheng and D. W. Gillies. Disk management for a hard real-time file system. In *8th Euromicro Workshop on Real-Time Systems*, page 0255, 1996.

[7] IBM. 146gb 15k 3.5" hot-swap SAS. http://www-132.ibm.com/webapp/wcs/stores/servlet/ProductDisplay?catalogId=-840&storeId=1&langId=-1&dualCurrId=73&categoryId=4611686018425093834&productId=4611686018425132252, 2007. [Online; accessed 04-October-2007].

[8] K. H. Kim, J. Y. Hwang, S. H. Lim, J. W. Cho, and K. H. Park. A real-time disk scheduler for multimedia integrated server considering the disk internal scheduler. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 124–130, Washington, DC, USA, 2003. IEEE Computer Society.

[9] F. Limited. MAX3147RC MAX3073RC MAX3036RC hard disk drives product/maintenance manual. http://193.128.183.41/home/v3_ftrack.asp?mtr=/support/disk/manuals/c141-e237-01en.pdf, 2005.

[10] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock scheduling outside of disk firmware. In *Proceedings of the Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002. USENIX.

[11] Maxtor. Atlas 10K V. http://www.darklab.rutgers.edu/MERCURY/t15/disk/pdf, 2004.

[12] J. Qian and A. I. A. Wang. A behind-the-scene story on applying cross-layer coordination to disks and raids. Technical Report TR-071015, Florida State University Department of Computer Science, Florida, Oct. 2007.

[13] A. L. N. Reddy and J. Wyllie. Disk scheduling in a multimedia I/O system. In *MULTIMEDIA '93: Proceedings of the first ACM international conference on Multimedia*, pages 225–233, New York, NY, USA, 1993. ACM Press.

[14] L. Reuther and M. Pohlack. Rotational-position-aware real-time disk scheduling using a dynamic active subset (DAS). In *RTSS '03: Proceedings of the 24th IEEE International Real-Time Systems Symposium*, page 374, Washington, DC, USA, 2003. IEEE Computer Society.

[15] P. J. Shenoy and H. M. Vin. Cello: a disk scheduling framework for next generation operating systems. In *SIGMETRICS '98/PERFORMANCE '98: Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 44–55, New York, NY, USA, 1998. ACM Press.

[16] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. In *SIGMETRICS '95/PERFORMANCE '95: Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, pages 146–156, New York, NY, USA, 1995. ACM Press.

[17] J. C. Wu and S. A. Brandt. Storage access support for soft real-time applications. In *RTAS '04: Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*, page 164, Washington, DC, USA, 2004. IEEE Computer Society.

[18] Y. Zhang and R. West. Process-aware interrupt scheduling and accounting. In *Proc. 27th Real Time Systems Symposium*, Rio de Janeiro, Brazil, Dec. 2006.