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

Many contemporary disk drives have built-in queues and schedulers. These features can improve I/O performance, by offloading work from the system's main processor, avoiding disk idle time, and taking advantage of vendor-specific disk characteristics. At the same time, they pose challenges for scheduling requests that have real-time requirements, since the operating system has less visibility and control over service times. While it may be possible for an operating system to obtain more predictable real-time performance by bypassing the on-disk queue and scheduler, the diversity and continuing evolution of disk drives make it difficult to extract the necessary detailed timing characteristics of a specific disk, and to generalize that approach to all hard drives.

This paper demonstrates three techniques we developed in the Linux operating system to bound real-time request response times for disks with internal queues and schedulers. The first technique 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 is to limit 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. These techniques can be generalized to disks from different vendors.

[*]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.

1 Introduction

The request service time of a disk drive is many orders of magnitude slower when compared to most other electronic components of a computer. To minimize the mechanical movements of the disk and improve its performance, one common optimization is to reorder requests. For many decades, the I/O scheduler component of an operating system (OS) has been responsible for providing request reordering to achieve high throughput and 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 which largely hides the detailed data layout and capabilities of the disk. For instance, common intuition suggests 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 popular perception that issuing requests with consecutive disk 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. This capability enables the disk head to 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 begins [16].

Given that the OS has limited knowledge of the layout, capabilities, timing characteristics, and the real-time state of individual disks, 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 OS and reorder them. Figure 1 il-

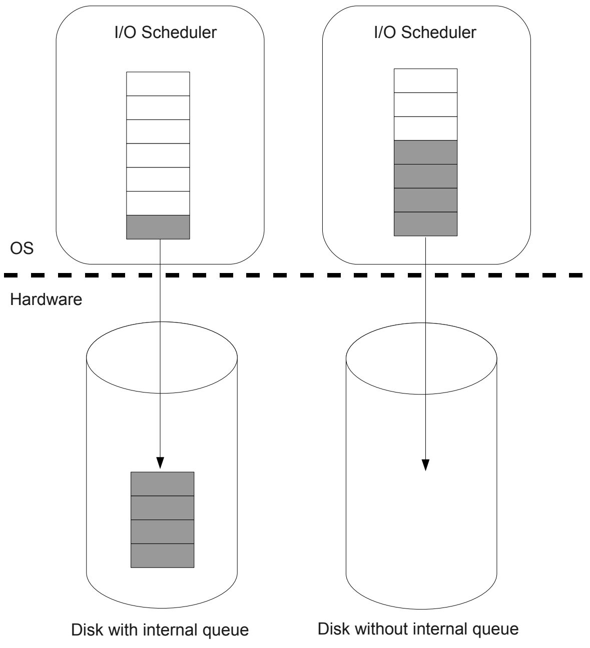

Figure 1. Disk-request queuing schemes.

lustrates the key difference between disks with an internal queue/scheduler and those without. Instead of requiring the OS to maintain all requests in the I/O scheduler framework, disks with a built-in queue allow the OS to issue multiple requests to the disk without waiting for the completion of a previous request. This permits multiple requests to be pending at the OS level as well as the hardware level. Reordering may occur at both levels, but once requests have been sent to the disk, control over their service order shifts from the OS 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 OS. In order for an OS to provide completion time guarantees, rather than just having to determine the completion time of one request at a time the OS needs to predict the order in which the disk will serve multiple requests. Further, the worst-case service time of requests sent to the disk can be increased to many times that of a disk that does not contain an internal queue. All these concerns need to be addressed in order to use these disks to serve real-time requests.

2 Motivation

Certain characteristics of disk drives make it difficult to predict I/O response times accurately. One such characteristic is the variability of service times caused by the state of the disk due to a prior request. With disks that allow just one outstanding request at a time, a new request sent to the 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 the 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 in the order of tens of milliseconds.

Many contemporary disks, including most SCSI and new SATA drives, have an additional state parameter in the form of an internal queue. To send multiple requests to the disk, the command queuing capability provides the protocol with three common policy variants: simple, ordered, and head-of-the-queue [1]. Policies are specified with a tag as each request is sent to the disk. The “simple” tag indicates that the request may be reordered with other requests also marked as “simple”. The “ordered” tag specifies that all older requests must be completed before the “ordered” request begins its operation. The “ordered” request will then be served followed by any remaining “ordered” or “simple” tagged commands. Finally, 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 both 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 disks with internal queues/schedulers, we conducted a number of simple experiments. 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 data sheets for a particular disk. 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 Fujitsu drive’s data sheet [9], the maximum seek time and rotational latency are 9 and 4

Hardware/software	Configurations
Processor	Pentium D 830, 3GHz, 2x16-KB L1 cache, 2x1-MB L2 cache
Memory	2-GB dual-channel DDR2 533
Hard disk controller	Adaptec 4805SAS
Hard disks	Maxtor ATLAS 10K V, 73-GB, 10,000 RPM, 8-MB on-disk cache, SAS (3Gbps) [11]
	Fujitsu MAX3036RC, 36.7-GB, 15,000 RPM, 16-MB on-disk cache, SAS (3Gbps) [9]
	IBM 40K1044, 146.8-GB, 15,000 RPM, 8-MB on-disk cache, SAS (3Gbps) [7]
Operating system	Linux 2.6.21-RT PREEMPT

Table 1. Hardware and software experimental specifications.

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. Consider a request of size 256-KB, as in our experiments. Since modern disks store more information on outer tracks than inner tracks, the span of such a request could range from half of a track to sometimes more than one track on various disks. Thus, a request could potentially take two rotations to access the data, resulting in a worst-case bound of 17 msec. Clearly, this crude estimation overlooks factors such as settling time, thermal recalibration, read errors, bad sectors, etc. However, the aim is to develop a back-of-the-envelope measurement on the range of expected service times.

To validate these estimated service times empirically, we created a task that issues 256-KB requests to random disk locations. Each request's completion time was 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 expected maximum 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. With the observed maximum completion time of 19 msec, using disks for some real-time applications seems quite plausible.

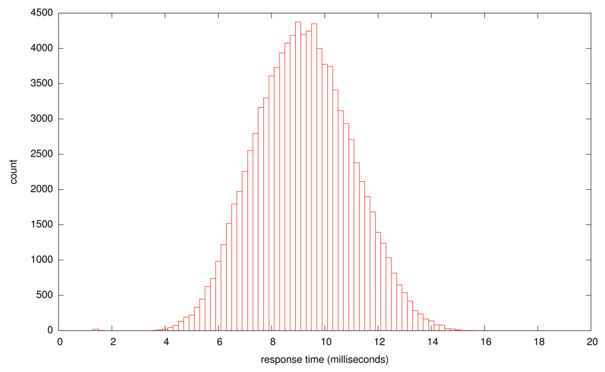

Figure 2. Observed disk completion times with no interference.

2.2 Handling Background Requests

The above bound does not consider environments with mixed workloads, where real-time requests (with deadlines) and best-effort requests coexist. This mixture increases the worst-case service completion time of the disk. On a disk that can accept just 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 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 scheduler is then able to create a service order to maximize the efficiency of the disk. This capability, however, poses a problem regarding requests with timing constraints, since meeting timing constraints can be at odds with servicing the given requests efficiently.

Consider a scenario with both real-time and best-effort requests. For real-time requests, the I/O scheduler at the device driver should send real-time requests to the disk ahead of best-effort requests. Otherwise, a backlog of best-effort requests could easily accumulate in the on-disk queue, causing high real-time I/O latencies.

In order to give priority to real-time requests the OS must be able to distinguish them. One way in which this can be done is through I/O priorities. For example, the Linux operating system call *ioprio_set()* allows setting the I/O class and priority of a process. Unfortunately, among the Linux disk schedulers, only the the "complete fairness queueing" (CFQ) scheduler pays any attention to these I/O scheduling attributes. While we did perform some experiments using the CFQ scheduler, it incurs additional complexity that hinders the understanding of disk service times, and generally performs poorly for tasks with deadlines. The results of those experiments are reported in Section 4.

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 honors the priorities set by the individual applications and does not merge requests. It issues the requests within individual priority levels in a first-come-first-served fashion. All disk-location-based sorting and merging are handled by the on-disk scheduler which frees the device-driver-level scheduler from making assumptions about the physical layout of logical sectors on the disk, eliminates the burden of ordering requests, and reduces CPU load.

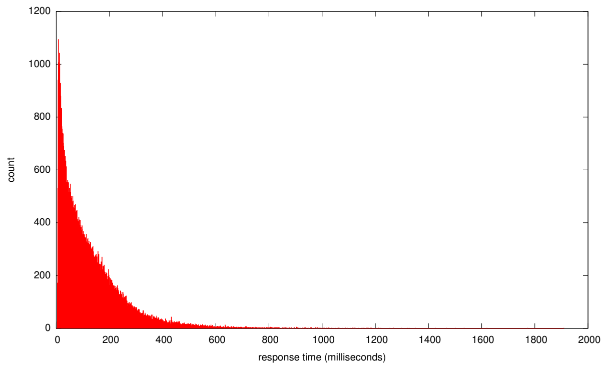

Figure 3. Completion times for real-time requests in the presence of background activity using the RTIO scheduler.

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 task is a best-effort multi-threaded process that generates up to 450 concurrent read requests continuously. The idea is to generate significant interference for the real-time task. The results for the completion times of the real-time requests using the Fujitsu drive are graphed in Figure 3. The reordering of requests by the disk can cause unexpectedly long response times. The largest observed latency was around 1.9 seconds. The best-effort 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 worst-case response time, these two goals are often in conflict.

Disks with an internal scheduler/queue can significantly increase the variance of I/O completion times and reduce their predictability as opposed to disks without. However, the internal queue does provide some benefits. Not only does it provide good throughput, 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 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 a way to address these issues, it is unlikely 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 of around two seconds. This maximum suggests that the disk is aware of starved requests, and it forces

[1]The maximum observed throughput of the disk on other experiments in which we fully loaded the disk with sequential read requests ranged from 73.0 to 94.7 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, rotational delay, and seek time for 20 requests. For our experiment, this is 256KB/(3 msec (to transfer 256KB) + 4 msec (per rotation)/2 + 11 msec (worst-case seek time)/20), giving a throughput of 46.1 MB/sec. This is not far from the 40 MB/sec achieved in our experiments.

these requests to be served even though they are not the most efficient ones to be served 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.

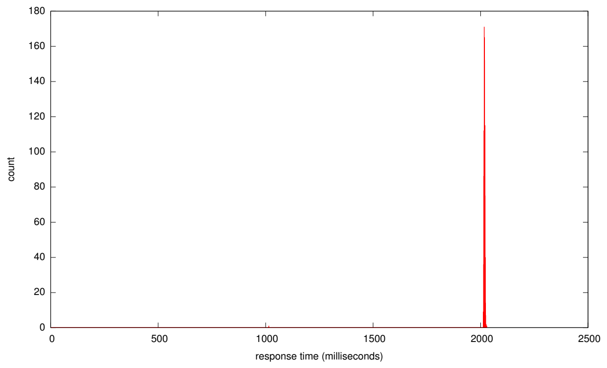

Figure 4. Disk's observed starvation prevention.

To be specific, 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 their 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 read response time of the disk. Given this information, real-time applications that do not require a response time less than 2.03 seconds do not need anything special to be done. The on-disk scheduling will provide best-effort throughput of 40 MB/sec, while preventing starved requests. Intriguingly, the spike is cleanly defined suggesting that this particular disk has a strong notion and control of completion times rather than simply counting requests before forcing a starved request to be served. Note that all disks do not have the same maximum starvation times. It is likely that most disk's do have some sort of built-in starvation mechanism, since not having one could lead to unexpectedly long delays at times. On the disks that we have tested, the maximum starvation time ranges from approximately 1.5 to 2.5 seconds.

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

The reordering of the requests queued on the disk is not the only cause of the extended completion times observable in Figure 3. As on-disk queue slots become available, newly arrived requests can potentially be served before the previously sent requests, leading to completion times greater than that of just servicing the number of possible requests permitted to be queued on a disk.

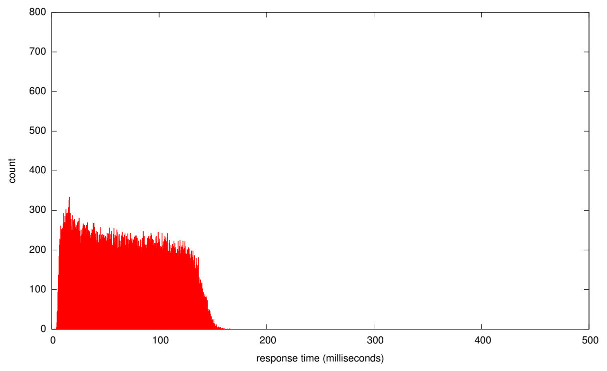

Figure 5. Effect of draining requests on the Fujitsu hard disk.

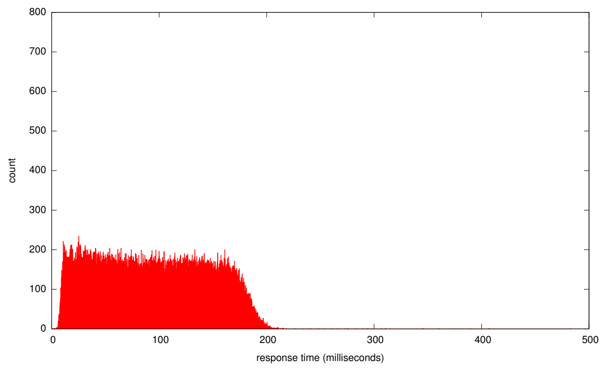

Figure 6. Effect of draining requests on the Maxtor hard disk.

To discover 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

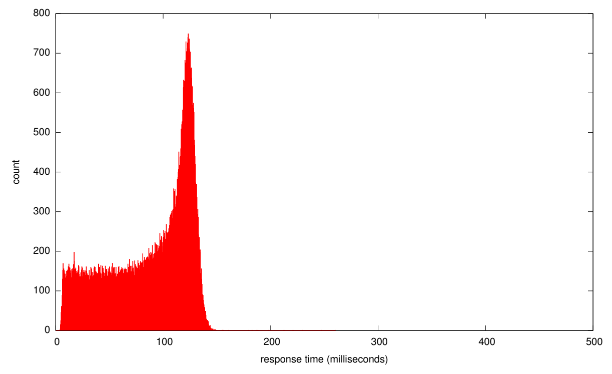

Figure 7. Effect of draining requests on the IBM hard disk.

requests to the disk. As can be seen in Figure 5, the completion times are significantly shorter than the worst cases shown in Figure 3. However, contrary to our intuition, real-time requests are more likely to be served with a shorter completion time, while it might seem that a real-time request should have had an equal chance of being chosen among all the requests to be served next. With an in-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 *n*th request to be served, as well as the probability of various requests being coalesced to be served with fewer rotations. In this case, 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.

While draining can prevent completion time constraints from being violated, this technique relies on knowing the worst-case drain time for a given number of outstanding requests on the 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 starvation. 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 inferring the scheduling algorithm is evident. Furthermore, 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.

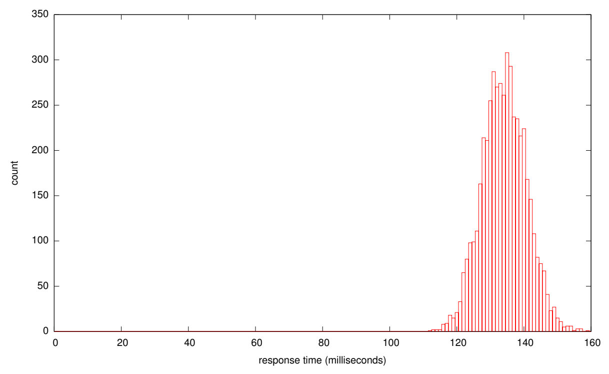

Figure 8. Empirically determining the drain time for 20 outstanding disk requests on the Fujitsu drive.

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 reasonably large requests to the disk with a uniformly random 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 20 256-KB requests on the Fujitsu drive is shown 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

To implement the proposed draining policy, we modified our RTIO scheduler, so that once a real-time request has been issued to the disk, RTIO stops issuing further requests. If no real-time requests are present, RTIO 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 reads 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 read requests with a maximum of 450 outstanding requests. Figure 9 shows that no completion times exceeded 160 msec, and no deadlines were missed. (This

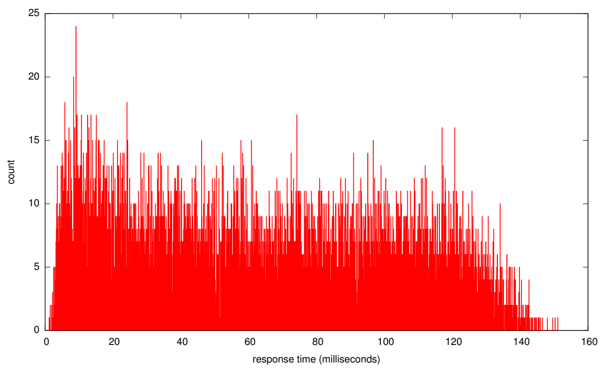

Figure 9. Draining the queue to preserve completion times of 160 msec.

is compared to the 2.03 second worst-case raw latency of the disk.) The throughput for best-effort requests 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, creating a more general case of draining. This mechanism artificially 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 (subject to the timing limitations of physical disks of course).

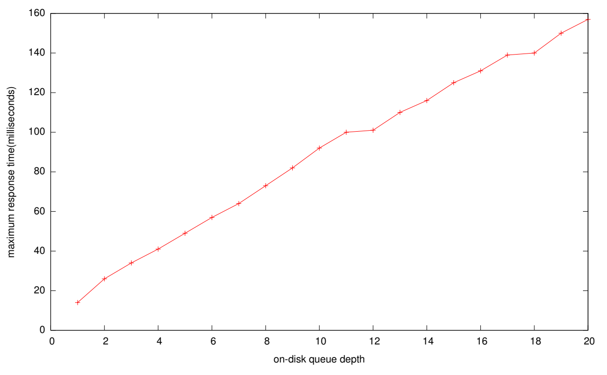

Figure 10. Maximum observed drain times for on-disk queue depths.

For instance, one can see from Figure 10 that to meet the completion time constraint of 75 msec using a queue depth of less than 9 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 grants the disk more flexibility when choosing requests to service, resulting in better scheduling decisions. Therefore, in this case, we would choose a queue depth of 8. The best-effort tasks must be limited to a queue depth of 7, with one slot reserved for the real-time request.

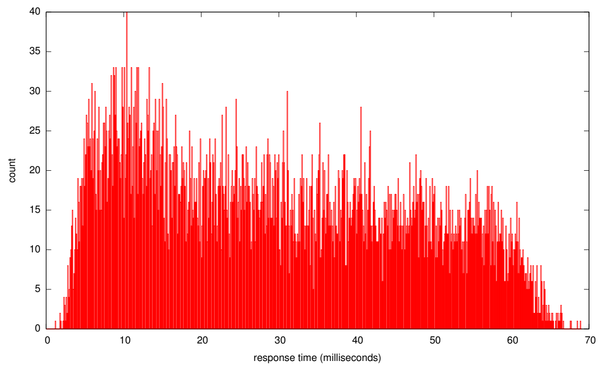

Figure 11. Limiting and draining the queue to preserve completion times of 75 msec.

To verify our tabulated queue length, a similar experiment was performed as before, changing only the period and deadline to be 75 msec. 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 60% decline 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 conducted experiments to compare the performance of RTIO with disk scheduling algorithms provided by the standard Linux kernel.

One of the standard Linux disk schedulers is called the “deadline scheduler”. It might seem that this scheduler should be effective in bounding I/O completion times, however, the name is misleading. First, the “deadline” in this case means the time before sending a request to the disk drive. Second, the so-called deadline scheduler does not support the use of I/O priorities, or more than one relative deadline, meaning that all real-time I/Os will be handled the same as non-real-time I/Os. Figure 12 shows the results of constantly sending 450 best-effort requests to the disk while at the same time recording the completion time of the real-time request being issued periodically. When comparing the results of the deadline scheduler with those of Figure 3 the importance of honoring I/O priorities becomes apparent.

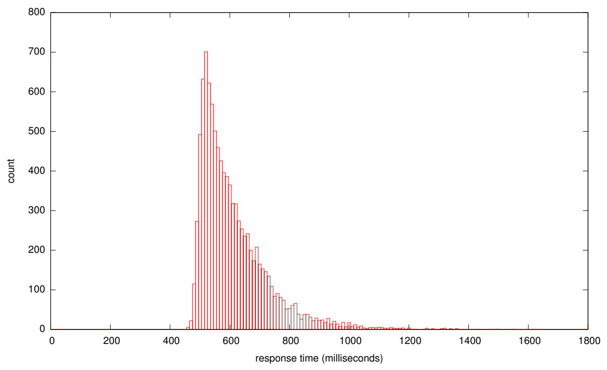

Figure 12. Completion times for real-time requests using the Linux deadline I/O scheduler.

While the so-called deadline scheduler in Linux could be modified to take advantage of the I/O priorities, the reordering problem of the disk's internal scheduler would still exist. That is, deadline scheduling is not just getting the requests to the disk in deadline order, but also getting the results back from the disk within a specified deadline. The deadline scheduler could be further modified to take advantage of "draining" by setting a deadline on requests that have yet to return from the disk as well as requests residing in the I/O scheduler's queue. When a request has been on the disk longer than a specified time, the draining mechanism should then be invoked. Draining would then last until no requests submitted to the disk have exceeded their deadlines. While this approach would not provide exact completion times for any given request, it would allow for much better deadline performance than seen currently.

We also studied the default CFQ I/O scheduler for Linux. CFQ 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 in a timely manner, not to mention the requests queued on disk. Given that the combined queue depth can be quite large, a real-time request may potentially be forced to wait for hundreds of requests to be completed.

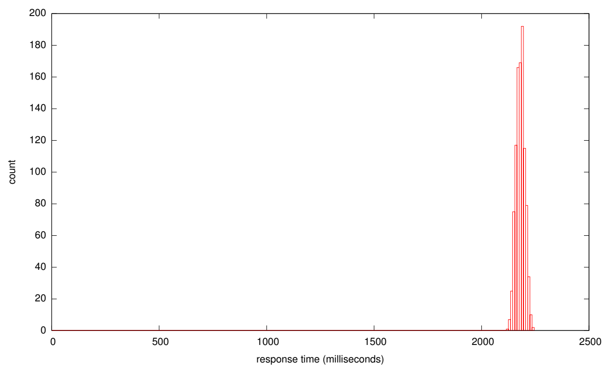

Figure 13. Completion times for real-time requests using the Linux CFQ I/O scheduler.

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 to those in Figure 3. These experiments provided a continuous backlog of 450 best-effort requests to random locations on disk, in addition to one real-time request sent periodically. The request size was again limited to 256-KB. The only difference was in the use of CFQ rather than RTIO.

Figure 13 shows that real-time completion times can exceed 2 seconds. Inspecting the kernel code in search of an explanation, we found that the real-time request handling is not correctly being implemented. When a new real-time request arrives, the CFQ scheduler does not always allow the newly issued real-time request to preempt preexisting requests in the driver's queue. In an attempt to alleviate this problem, we made a minor modification to the code to force the newly-issued real-time request to preempt all best-effort requests residing in the driver queue. This change, however, had the side effect of allowing only one request on the disk at a time. Although this change yielded low completions times with a maximum of 17 msec, the best-effort throughput degraded to 18 MB/sec.

Further understanding of the CFQ code showed that the low throughput may be attributed to how CFQ treats each thread as a first-class scheduling entity. Because our experiments used numerous threads performing blocking I/Os, and since CFQ introduces artificial delays and anticipates that each thread will issue additional I/Os near the current requests location, our multi-threaded best-effort process can no longer result in multiple requests being sent to the on-disk queue. It might seem that asynchronous requests could be used; however, the current Linux C library only emulates the asynchronous mechanism by spawning worker threads that perform blocking requests. Further, the maximum number of worker threads is limited to 20. Our final solution was to modify the CFQ code to schedule according to two groups of requests: one real-time and one best-effort. This modified CFQ now can forward multiple requests to the on-disk queue.

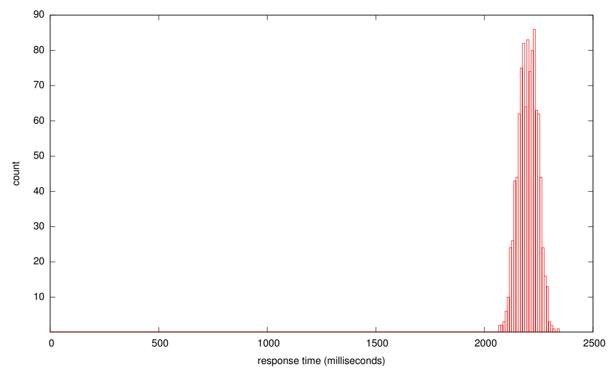

Figure 14. Completion times for real-time requests using a modified Linux CFQ I/O scheduler.

After the modifications, the real-time completion times are shown in Figure 14. Without the limit imposed by the disk's starvation control algorithm, the real-time response might have been even worse. The real-time performance is still poor 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 completion times for the majority of real-time requests are over 2 seconds, considering the requests are randomly distributed across the entire disk, and real-time requests should have a good chance of not being chosen last. This problem occurs because CFQ sorts the random collection of best-effort requests, resulting in a stream of arriving requests that are more likely to be closer to the current disk head location than to the real-time request. Therefore, servicing the best-effort requests prior to the real-time request is more efficient. With the combination of merging and sorting, CFQ issues near-sequential reads for the best-effort tasks, and can achieve throughput of 51 MB/sec. However, the cost is severe degradation of real-time performance.

5 Multiple Real-Time Requests

In this section we evaluate how our approach extends to handle multiple real-time disk requests. We provide only preliminary results and leave further analysis to future work.

Draining ensures that the disk has sufficient time to service all requests currently residing on the disk's queue without violating a real-time request that has been issued to the disk. To allow for multiple real-time requests, this strategy can be extended by ensuring enough time exists to service all requests on the disk before the earliest real-time request's deadline. This gives the advantage that draining for one request will also assist other real-time requests in meeting their deadlines.

A potential pitfall occurs when a second real-time request must be issued to the disk to ensure meeting of its deadline. However, sending the request to the disk will jeopardize the deadline of an already existing request on the disk. To overcome this problem we allow real-time request to reserve slots on the disk's queue. Reserving a slot means that the scheduler will assume that a real-time request exists on the disk whether or not it is present. This allows the draining of the previous scenario with 2 real-time requests to be alleviated. Even if the second real-time request gets serviced before the first real-time request we will have accounted for this case and should not miss any deadlines.

To evaluate this approach, we chose an experiment similar to that of Figure 11. The difference being that we will use, an arbitrarily chosen, 4 real-time requests instead of 1. Having 4 real-time tasks requires the reservation of 4 slots for the real-time requests and leaves 4 slots for best-effort requests. All real-time requests will have a period and deadline of 75 msec. The real-time tasks are offset from one and other by 18.75 msec. That is, after the first real-time task is started, the next task is started 18.75 msec later, the third at 37.5 msec and so on. This reduces the likelihood that all real-time requests will drain together and potentially increases interference. The completion times of real-time requests are shown in Figure 15. The throughput of the best-effort requests was recorded to be 26 MB/sec. All real-time requests completed within their deadline constraints, suggesting that our draining approach does generalize to multiple real-time requests. Given that the longest response time was only 54 msec, implies that more aggressive issuing of requests to increase best-effort throughput may be possible.

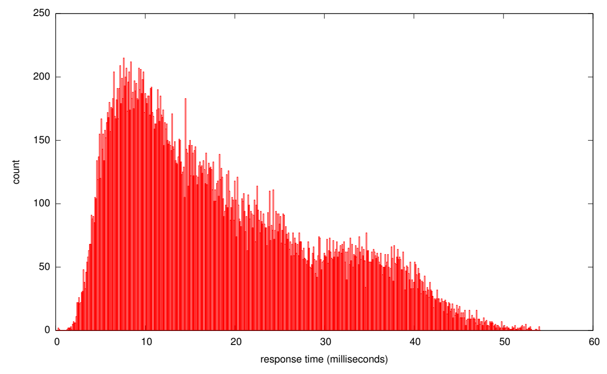

Figure 15. Completion times for four real-time requests.

6 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 increasingly difficult to obtain with the rapid growth in complexity and evolution of disk drives.

Reuther and Pohlack [14] and Lumb, Schindler and Ganger [10] 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 track of the disk's state accurately. 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 accurately keeping 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 a risk of leaving the disk idle while it waits for the next request from the CPU, thereby affecting 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, which can be highly vendor specific. Even if the device driver is run at maximum priority, an off-disk scheduler will have (1) less complete and less timely information, (2) a reduced amount of control due to the limited information provided by the disk I/O interface, and (3) 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 scheduler 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 its 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, it does not provide precise a priori completion time guarantees.

7 Conclusion

In this paper, we discussed how to use a hard disk's internal queue and scheduler without jeopardizing completion time constraints for real-time requests. While this goal may also be achieved by doing disk scheduling within the operating system, allowing the disk to make scheduling decisions offloads the device-driver-level I/O scheduling and permits optimizations that cannot be easily achieved in the operating system. The approaches we explored 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 view of its near-future requests while reordering requests to realize efficient use of the disk's resource. Additionally, the disk can achieve a higher level of concurrency with CPU processing, servicing on-disk requests without immediate attention from the operating system. These approaches further allow high-priority real-time processes to use the CPU with little impact on disk performance. The techniques in this paper can be applied on many different disk drives and are not vendor specific.

8 Acknowledgments

The authors would like to thank Mark Salyzyn of Adaptec for helping us bypass some of the SAS controller features. We are also thankful to the anonymous reviewers of the RTAS 2008 program committee for their suggestions and guidance.

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.