CSched : **Real-time disk scheduling with concurrent I/O request**

Carl Staelin, Gidi Amir, David Ben-Ovadia, Ram Dagan, Michael Melamed, Dave Staas

HP Laboratories
HPL-2011-11

Keyword(s):
Real-time disk scheduling, storage systems

Abstract:
We present a new real-time disk scheduling algorithm, *Concurrent Scheduler* or *CSched*, which maximizes throughput for modern storage devices while providing real-time access guarantees, with computational costs of $O(\log n)$. To maximize performance it ensures request concurrency at the device and maximizes the depth of a new Limited Cyclical SCAN (L-CSCAN) queue that optimizes the request sequence sent to the device. For realtime requests there is an additional SCAN-EDF queue in front of the L-CSCAN queue to absorb bursts of realtime requests until they can be drained to the L-CSCAN queue. The real-time guarantees are provided by managing the worst-case latency at each stage of the pipeline: SCAN-EDF, L-CSCAN, and device. CSched is configured by the tuple $\{\lambda, \sigma, \delta, \tau(r), N\}$, where λ and σ are the minimal initial slack time and workload burstiness and are properties of the workload, and where δ, $\tau(r)$, and N are the deviceworst-case latency, worst-case throughput rate time for a request, and maximal number of concurrent requests, and are experimentally determined properties of the storage device. An experimental evaluation of CSched shows that given sufficient initial slack time, the system throughput performance costs of providing real-time guarantees are negligible.

External Posting Date: January 21, 2011 [Fulltext]
Internal Posting Date: January 21, 2011 [Fulltext]

Approved for External Publication

CSched : Real-time disk scheduling with concurrent I/O requests

Carl Staelin, Gidi Amir, David Ben-Ovadia, Ram Dagan, Michael Melamed, and Dave Staas
Hewlett-Packard

Abstract

We present a new real-time disk scheduling algorithm, *Concurrent Scheduler* or *CSched*, which maximizes throughput for modern storage devices while providing real-time access guarantees, with computational costs of $O(\log n)$. To maximize performance it ensures request concurrency at the device and maximizes the depth of a new Limited Cyclical SCAN (L-CSCAN) queue that optimizes the request sequence sent to the device. For real-time requests there is an additional SCAN-EDF queue in front of the L-CSCAN queue to absorb bursts of real-time requests until they can be drained to the L-CSCAN queue. The real-time guarantees are provided by managing the worst-case latency at each stage of the pipeline: SCAN-EDF, L-CSCAN, and device. CSched is configured by the tuple $\{\lambda, \sigma, \delta, \tau(r), N\}$, where λ and σ are the minimal initial slack time and workload burstiness and are properties of the workload, and where δ, $\tau(r)$, and N are the device worst-case latency, worst-case throughput rate time for a request, and maximal number of concurrent requests, and are experimentally determined properties of the storage device. An experimental evaluation of CSched shows that given sufficient initial slack time, the system throughput performance costs of providing real-time guarantees are negligible.

1 Introduction

CSched is a real-time disk scheduling algorithm for mixed real-time and non-real-time workloads that is designed to provide performance similar to non-real-time disk schedulers. It does this by maximizing disk request concurrency; the system will never reduce the number of concurrent outstanding requests at the storage device in order to satisfy a real-time deadline. It uses a variant of the simple and efficient scheduler, Cyclical Scan (CSCAN) [34], to provide performance as close to optimal as possible while still providing real-time guarantees.

Real-time disk scheduling is an old and important topic, but most work assumes that it is scheduling for a single disk, and that it is the only scheduler in the path. However, most large systems no longer use solitary disks, but rather arrays of disks, usually with some data redundancy, so storage performance increases as you allow more concurrent requests, particularly with small requests. Also, nearly all disk devices incorporate an intelligent positional-aware disk scheduling algorithm so in many instances the disk itself may do more intelligent and efficient scheduling than external schedulers. Finally, disk scheduling may be present in multiple locations in the hardware, such as the disk itself and the RAID controller.

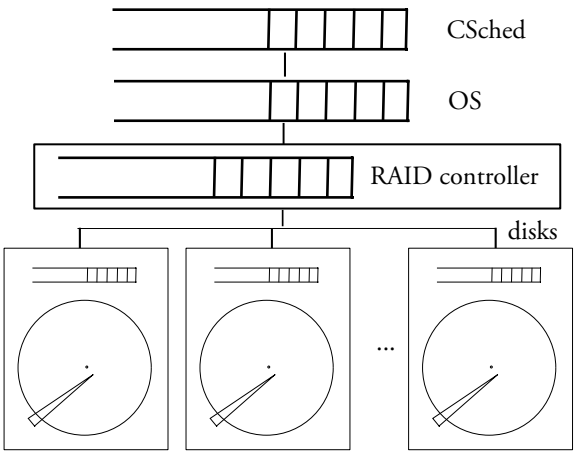

Figure 1: Multi-level scheduler.

As shown in Figure 1, we are building a multi-level scheduling solution. The application can only submit disk requests to the operating system; it cannot influence or control any of the scheduling decisions taken at the other schedulers in the operating system, RAID controller, and disk. In particular, the application must

assume that: (1) requests are non-preemptible, (2) concurrent requests may be serviced in any order, and (3) the operating system and device schedulers may not be fair and may starve requests. In addition, the application's scheduler should submit requests to the operating system in a way that allows the lower-level schedulers to optimize performance, while still providing the soft real-time performance guarantees.

From a performance standpoint, the intra-device queues are responsible for the more sophisticated rotational positional aware scheduling within the relatively small set of concurrent requests. The job of the application scheduler is to try and cluster requests as closely in space and time so that the low-level, intra-device schedulers have a richer set of optimization possibilities because at any given point in time, the small set of concurrent requests are clustered tightly enough that the rotational positional aware scheduling can make a difference.

We define *throughput rate time* as the average time between request completions, which may be defined as the inverse of the IOPS. Throughput rate times usually depend on at least the request size, whether the request is sequential or random, whether it is a read or write operation, and the number of concurrent requests allowed at the device. We define worst-case throughput rate time for a request as $\tau(r)$, which is the throughput rate time with full N concurrency for a fully random request stream. As described in Section 3, $\tau(r)$ is measured experimentally offline. We use this value instead of the average service time of a request because of the concurrency of the system. Using throughput rate time as the expected resource requirement for a request allows simpler and more accurate accounting of expected device utilization.

The results of various performance experiments are presented throughout the paper. These results were generated on an HP DL380 server with two quad-core 2.93GHz Intel Xeon X5570 processors and 8GB of RAM running Windows Server 2008 64-bit. The storage array used for the performance testing was built from eight 146GB 15k RPM SAS disks, attached on two SCSI cables (four disks on each cable) to an HP P410i storage controller. The storage was configured into a RAID0 array with 128kB stripes and a total usable size of 1.1TB. The software accessed the raw device directly, rather than using a file system, and it utilized threading and Windows' asynchronous I/O capabilities to send multiple I/Os to the device at once.

Unless otherwise stated, the benchmarks used a closed-loop environment with uniform random disk addresses. Usually there were four thousand (4,000) best effort read requests in the queue waiting to be sent to the device, and the device had sixty (60) concurrent requests. Real-time requests usually had an initial slack time of thirty (30) seconds, and used a uniform random distribution of disk addresses. Their deadlines were uniformly spaced through time to match the desired throughput rate given the request size. New real-time requests were added to the queue when needed to ensure the thirty second initial slack time, regardless of whether or how previous real-time requests had already been serviced.

2 Prior Work

There is a rich body of work on disk scheduling algorithms for non real-time applications and environments. A classic algorithm which requires minimal complexity and provides acceptable performance is the SCAN [6], or elevator algorithm, also sometimes called LOOK. In this case, pending requests are sorted according to disk offset and are submitted to disk in order. When the algorithm reaches the end of the disk, it reverses direction. A better variant of this algorithm is Cyclical SCAN or CSCAN [34], and a variant called Cyclical LOOK or C-LOOK [22]. In this case, when there are no more requests in the current direction of travel, it jumps to the first request and begins again. This approach has better performance than SCAN and may also result in better fairness. N-Step SCAN [7] is a variant of SCAN or CSCAN designed to improve fairness, or rather to limit unfairness, whereby the request stream is divided into N-request sized chunks, and SCAN is used within each chunk.

However, these early algorithms overlook one aspect of modern disk drives, namely that rotation delay may be dominant compared to seek time for short seek distances. A variety of algorithms, such as shortest time first (STF), grouped shortest time first (GSTF), and aged shortest time first (ASTF) [36, 16, 45], attempt to improve disk utilization and throughput by taking into account rotational position as well as seek position, resulting in generally improved utilization. Pure STF tends to suffer from starvation and long maximal service times. GSTF and ASTF attempt to reduce these issues by forcing STF to occasionally "jump" from one area of the disk to another, which likely also has a higher density of waiting requests. Later innovations also took into account on-disk caching and pre-fetching, yielding further performance improvements in some cases [45].

Starting in the early 1990's, SCSI disks [1], and later ATA disks [15, 5], supported multiple concurrent disk requests utilizing positional aware disk scheduling algorithms in the disk to provide higher performance. In combination with the widespread adoption of RAID devices, device-level support for concurrent I/O requests means that it is generally necessary to send multiple concurrent requests to storage devices to obtain optimal performance. This shift to concurrent I/Os and intra-device scheduling means that even physical clients of storage ar-

rays have at least a three-level scheduling solution: client operating system, controller, and disk. Additionally, with the advent and popular adoption of virtualization technologies, the single operating system scheduler in the client may be replaced with the scheduler in the virtual client OS and potentially a second scheduler in the virtual monitor. This layering of schedulers may result in unexpected and undesired scheduling behavior [47].

As an aside, this trend towards virtualization and shared resources makes real-time scheduling difficult or impossible, as accurate worst-case service estimation is non-trivial when the low-level device may behave unpredictably due to competing requests from other systems and the lack of any priority system at the hardware interconnect layers. Unless or until such functionality is added to the standard I/O interfaces, real-time systems must use dedicated hardware that behaves in a reasonably predictable fashion. However, there has been recent work on improving fairness and performance isolation for such shared storage servers, such as [35, 8, 11, 12, 18]

Typically, real-time systems avoid using disks, because of the variability in disk access latencies. However, with the advent of applications such as video-on-demand in the early 1990s, with soft real-time constraints and massive data storage requirements, real-time disk scheduling algorithms became important, and research in this area blossomed over the last two decades. A wide variety of real-time disk scheduling algorithms have been developed, largely for these multimedia storage servers [29, 24, 14, 2, 4, 25, 3, 38, 13, 30, 19, 27, 39, 48].

A simple algorithm is earliest deadline first (EDF) [21], where requests are processed according to the deadline order, from earliest to latest. However, this algorithm is known to suffer from poor disk performance because it makes no attempt to minimize seek and rotational delays [29]. SCAN-EDF [29] is a variant developed for multimedia systems, where large batches of I/Os are submitted periodically, so many I/Os have the same completion deadline. The variation is that all I/Os with the same completion deadline are processed in SCAN order, yielding far higher performance than EDF in this application.

One approach used by a variety of real-time disk scheduling algorithms, such as [4, 3, 25, 2, 19] is to build an initial EDF-based schedule, and to then revise that schedule to optimize latencies (typically just seek delays) while still meeting the deadline requirements. For example, one approach is to sequentially find the maximal group of EDF requests whose requests may be scheduled in SCAN order [2]. Some variants can also incorporate or intersperse best-effort requests in the request stream, such as RG-SCAN [3]. Related algorithms include MS-EDF [13], which uses a branch-and-bound search strategy to find a minimal seek cost disk schedule. However, all these approaches suffer from the limitation that they only schedule a single request at a time to disk, which limits performance on multi-disk RAID systems by ignoring available parallelism and on single-disk systems by not leveraging the disk's internal rotational position-aware scheduler which may further improve performance. In addition, these algorithms are typically computationally expensive.

DS-SCAN [10] manages a mix of real-time and best-effort requests by ensuring that real-time requests are issued in time to meet their deadlines but otherwise uses efficient non-real-time scheduling. It is a combination of earliest deadline first (EDF) real-time disk scheduling and CSCAN disk scheduling. When real-time requests are not in danger of missing their deadlines, both real-time and best-effort requests are passed to storage one at a time using the CSCAN algorithm. However, when submitting a request might cause a real-time to miss its deadline, DS-SCAN submits the nearest-deadline-first request, regardless of its "position" on the disk. DS-SCAN is similar in effect, although not architecture, to the "slack-stealing" solution developed as part of RT-Mach [24]. CDS-SCAN [37] extends DS-SCAN to allow and account for concurrent outstanding requests at the device, yielding significant performance improvements when using RAID devices.

Nearly all the papers assume that the worst-case service time must be used for all requests when analyzing or managing a stream of real-time requests, meaning that the system assume that every single request in a stream will (or may) perform in a worst-case fashion. In contrast, Fahrrad [27] assumes that a batch of requests will be served in the average service time, and the worst case assumption is $N \cdot \bar{s} + s_{worst}$ where N is the number of requests in the batch, $\bar{s}$ is the average service time for a request, and s_{worst} is the worst case service time for a request. However, Fahrrad also has the assumption that in a given batch it may cancel requests (not submit them to the disk) if the system is running behind and sending the next request to the device may cause the batch to use more than its allotted worst-case time estimate. For example, if at request k, $k < N, N \cdot \bar{s} < \sum_{i=1}^{k} s_i$, then the remaining requests in the batch are postponed to the next period.

Only one paper [14] appears to investigate concurrent I/O with a real-time scheduler, EDF [21] and SCAN-EDF [29], and found that performance improved when allowing a second request to be overlapped with the first request. In this paper most of the gain was obtained by overlapping execution of different portions of the I/O service, such as SCSI protocol overheads for the next request overlapped with the seek, rotational, and data transfer delays of the previous request.

One recent paper [41] appears to improve performance by briefly idling the disk after completing synchronous

disk requests, in the expectation that an application will quickly submit a request for the next block.

*p*Clock [12] is the most similar to our proposed solution, except that it allows itself the freedom to revise deadlines if a real-time stream uses more allocated bandwidth. It allows concurrent requests to be sent to the device, but it assumes that requests will finish within a small ammount of time. It characterizes workflows using 3-tuples: (σ, ρ, λ), where σ is the maximum burst size (number of I/Os), ρ is the arrival rate (in I/Os per second or IOPS), and λ is an upper bound on the latency of an I/O request (in ms). *p*Clock uses a *leaky bucket* model [26, 32] with the parameters σ and ρ above to describe and limit a real-time stream's request submission rate and burstiness. Briefly, the leaky bucket model may be described as follows. The *arrival function* for a request stream, $R(s,t)$ is the total number of I/O requests man in the time interval $[s,t]$. A request stream is *well behaved* if $R(s,t) \leq \sigma + \rho(t-s)$ for all time intervals $[s,t]$.

Empiric disk modeling and characterization has been done for years [31]. Often such models assume some general knowledge of the internal workings and data layout of the disk drive, sometimes with more detailed knowledge discovered by querying the device [46, 33]. However, aggregated storage devices, such as disk arrays, are far more complex, so it is difficult or impossible to develop accurate low-level models of all the parts [43]. One approach has been to treat the prediction of request services times and device throughput as a machine learning prediction problem [17, 23, 42, 40, 44]. In general, the models can accurately predict *average* response time and throughput as a function of device load (number of pending or concurrent requests) and workload (random versus sequential).

3 Device Characterization

In order to provide real-time guarantees, we must first characterize the device. There are a number of parameters that may impact the device performance, such as concurrency, request size, and request sequentiality. Accurately characterizing the device is complicated by the fact that there are potentially several devices interacting in interesting ways, such as the disk scheduler within the operating system, the disk scheduler and write cache in the RAID controller, and the disk scheduler in the disk itself.

Once a real-time request has been sent to the device, the application may not cancel or interrupt the request. The only way to ensure that outstanding real-time requests complete in time is to either: (1) submit real-time requests far enough in advance so that they always complete in time, or (2) not send any more requests to the device when a real-time request is in danger of missing its deadline. Option (2) starves the intra-device schedulers of alternative requests to service and forces it to service the threatened request. As we shall see, it is vital that the concurrency be maximized at all times or else performance suffers. This means that we must submit real-time requests to the device at least the worst-case service time in advance of their deadlines so that they will complete in time.

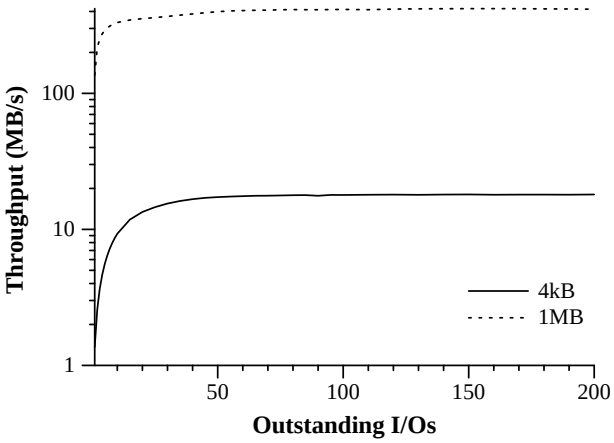

Figure 2: Throughput vs. Outstanding I/Os

Figure 2 shows throughput as a function of request concurrency at the device for two representative request sizes, 4kB and 1MB. The benchmark was done using a closed-loop system with four thousand (4,000) requests using a uniform random distribution of disk addresses, and passed through a CSCAN queue. Clearly there is significant performance benefit to using concurrent requests, but the benefits seems to reach the knee of the curve by about sixty (60) concurrent requests. Since this is also about the maximal concurrency available in the SCSI protocol, we use this as the default concurrency for our device in the rest of the experiments.

Figure 3 shows throughput as a function of the request size when there is a single request at a time at the device and when there are sixty concurrent requests at a time. This curve demonstrates that the benefits of utilizing the available concurrency are consistently significant across the whole range of request sizes, and not just a few sizes.

Figure 4 demonstrates how performance changes as a function of the size of the CSCAN queue size. The larger the CSCAN queue is, the better job the CSCAN queue can do of clustering requests in space and time. The more tightly clustered requests are when they are sent to the disk, the better job the rotational positional aware scheduler may do to service multiple requests in a single rotation [9]. In fact, if you look at the two curves with sixty outstanding requests in Figure 4, you can see that as the CSCAN queue size increases, so does disk

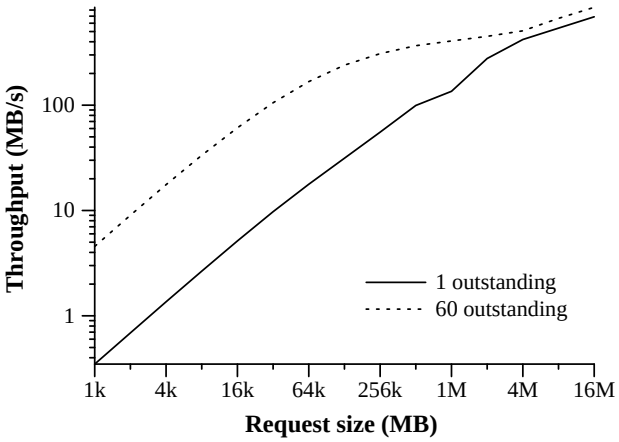

Figure 3: Throughput vs. Request size

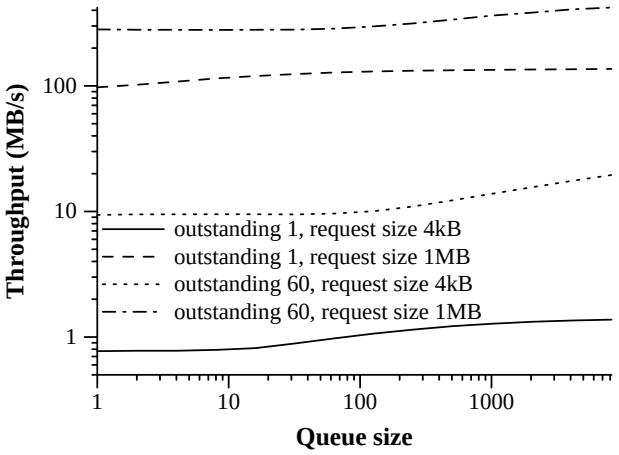

Figure 4: Throughput vs. CSCAN queue size

throughput.

Clearly, it is important to *both* maintain concurrency and maximize CSCAN queue size to maximize throughput. For this device, and likely for most disk arrays, throughput increases as a function of concurrency and CSCAN queue size. Concurrency, up to about sixty requests, is more important than CSCAN queue size, so maximizing concurrency should be prioritized over maximizing CSCAN queue size.

It is important that performance not degrade as load increases, because otherwise the system can get into a negative feedback cycle where increased load causes degraded performance which causes further load increases. So long as the algorithm is able to maximize concurrency, or at least maintain maximal concurrency under load, and so long as increased load yields increasing, or non-decreasing, CSCAN queue sizes, performance will not decrease under increased load.

Assuming we are given a device which we will control using CSched, how do we characterize that device? We must measure the device's worst-case latency and throughput rate time for random and sequential request streams and store them in a configuration file. Calculating or determining worst-case service latency becomes much more interesting as soon as there are multiple outstanding requests at the disk, as there are a variety of new aspects that must be taken into account. For example, there is the queuing algorithm in the disk itself, which may be an algorithm such as SATF, which is not "fair" and may starve requests [16]. Similarly, if there are N concurrent requests outstanding at the disk, how does one compute the worst-case service time for those requests, especially if the requests were sent to the disk over time and so they do not have identical start times? To make matters even more difficult, workloads may have a mixture of request sizes, so how is that taken into account? The short answer is that we avoid all those issues by measuring the worst-case latency with full concurrency, across the spectrum of request sizes, and across the range of CSCAN queue sizes, and choose the maximal value.

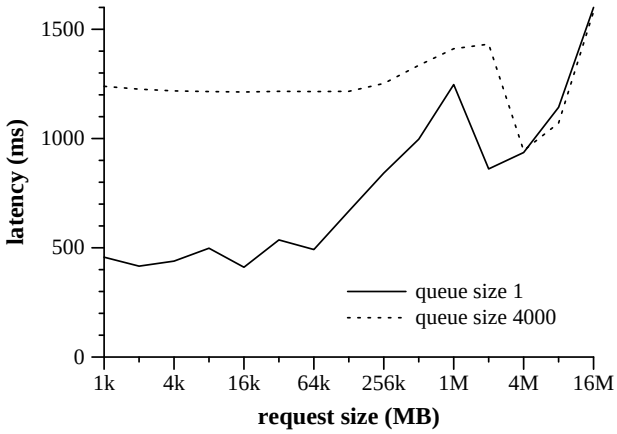

Figure 5: Worst-case service time vs. request size

Figure 5 shows the worst-case latency as a function of request size, with a purely random request stream for sixty outstanding requests, with both a large 4,000 request CSCAN queue before the disk, and without any queue before the disk. One interesting observation is that the worst-case time is shorter when the requests are given in random order (CSCAN queue length 1). This suggests that perhaps the lower-level schedulers are using an unfair scheduler such as SATF or ASATF [16, 45] and since the CSCAN does such a good job of clustering requests spatially, once the head moves away from a request, that request can starve. A second observation that the worst-case service time for 60 outstanding requests and a CSCAN queue of 4,000 requests is relatively stable at about 1,100ms across nearly the whole range of request sizes, suggesting that the lower-level scheduler has

some sort of anti-starvation parameter with a threshold or aging parameter set to about that value. Since worst-case performance is generally much worse for larger CSCAN queue sizes, and since it is relatively stable across request sizes, approximating the worst-case latency by the maximal value, about 1.6 seconds, generally doesn't overestimate the worst-case by much.

The worst-case throughput rate time $\tau(r)$ is measured using an experiment similar to that in Figure 2, except with an empty CSCAN queue so that the device is presented with a uniform random request stream. The configuration file simply stores the worst-case latency with full concurrency and the worst-case throughput as a function of request size.

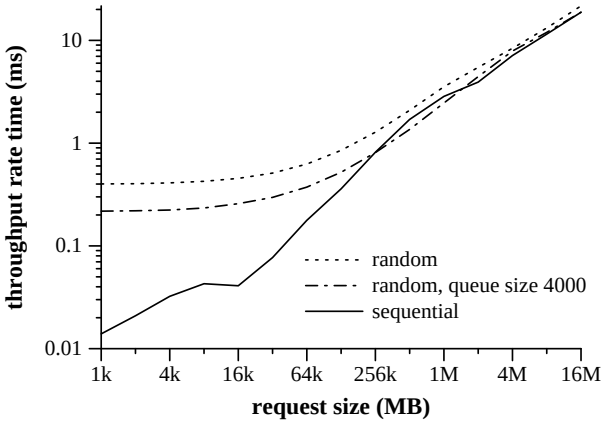

Figure 6: Throughput rate time vs. request size, for random requests

Figure 6 shows the throughput rate time as a function of request size for various request streams with sixty (60) concurrent requests at the device. The worst-case throughput rate time is the random request stream, because the request stream, as submitted to the device, is maximally randomized. For comparison, we also show the throughput rate time when the request stream is passed through a CSCAN queue that holds 4,000 requests, and the improved reference locality does improve performance, especially for smaller requests where the seek and rotational delay overheads dominate the total transfer costs. Performance for sequential request streams is by far the best.

We compute $\tau(r)$ using a set of lookup tables derived from the data in Figure 6. There is one table each for random and sequential requests, holding $\tau(r)$ for various request sizes. When necessary, we linearly interpolate between measured values.

One note is that RAID0 does not incur any additional penalty or overhead for write operations compared to read operations, so the models here do not include separate values for read and write operations, just sequential and random operations. If the storage system is using RAID5 or some other configuration where there is an asymmetry between read and write operation costs, then the characterization should measure the throughput rate time for the cross product of read vs. write, and random vs. sequential configurations.

Additionally, some larger and more modern storage systems automatically manage the placement of storage, so it is difficult or impossible to ensure that the actual physical disks holding the data for the logical disk, usually called a LUN, that is being used for the real-time data transactions, does not have other conflicting traffic from other LUNs that happen to be held on the same physical disks. In this case, probably the best option, if it is available, is to use some algorithm or solution such as *p*Clock [12] to manage or control the data rates available to each LUN and configure CSched for those guaranteed worst-case rates. If that is not possible, then probably the best option is to characterize the LUN when other LUNs are under load, and then add a fudge factor to the measured values worst-case service time and throughput rate times to provide additional protection against resource contention caused performance reductions. It may also be possible or desireable to dynamically track the observed throughput rate times to dynamically control the system in response to the current activity and performance of the storage device.

4 Architecture

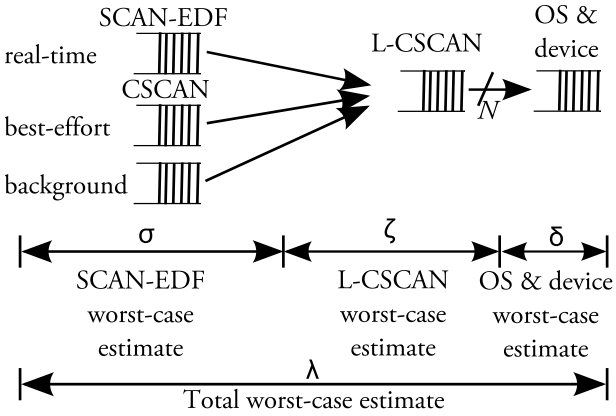

Figure 7: CSched architecture

The architecture of CSched is shown in Figure 7. Just before the device, with its internal black-box queueing, there is an L-CSCAN queue, which provides worst-case queueing delay guarantees, and which may contain real-time, best-effort, and background requests that are sorted in CSCAN order to maximize performance. To help limit the worst-case queue delay, the L-CSCAN queue size is

bounded, so new requests may have to wait to enter the L-CSCAN queue. To the left of the L-CSCAN queue, there is a SCAN-EDF queue for the real-time requests which are also waiting to be admitted to the L-CSCAN queue, and CSCAN queues for the best effort and background requests that cannot yet be admitted to the L-CSCAN queue.

The CSCAN, L-CSCAN, and SCAN-EDF algorithms are discussed in more detail in Sections 5, 6, and 7.1 respectively.

Figure 7 also suggests how one must calculate and control the worst-case behavior of the various elements to ensure that the real-time deadlines are met. Since the system is configured as a pipeline, each element of the pipeline may be considered independently: SCAN-EDF, L-CSCAN, and OS & device.

Most of the configuration parameters are shown in Figure 7. λ is the smallest initial slack time for real-time requests, and is a workload characteristic given to CSched. σ is the burstiness of the workload as computed using a *leaky bucket* model as in [26, 32, 12], except that instead of being specified in terms of requests it is specified in terms of time. Similar to [12] we define the *arrival function* of a request stream to be $R(s,t)$ and it is the sum of the worst-case throughput rate times $\tau(r)$ for all requests made in the time interval $[s,t]$, or $R(s,t) = \sum_{r_i \in (s,t)} \tau(r_i)$. We define $\sigma = \max_{s,t}\{R(s,t) - (t-s)\}$ for all time intervals $[s,t]$. δ is the worst-case device latency, and is measured experimentally offline, along with $\tau(r)$. N is the maximum number of concurrent requests allowed at the device. ζ is a derived configuration parameter that governs the maximal delay allowed in the L-CSCAN queue.

Please note that this is a general architecture which may be modified to fit other requirements. For example, it is possible to replace the single best-effort CSCAN queue with a Fahhrad-like queueing system [27] to ensure fairness of best-effort request streams across multiple clients.

The central element of the architecture is the L-CSCAN queue, which is a fair variant of the mostly-fair CSCAN with bounded worst-case queue delays. Since we need to ensure that all real-time requests enter the L-CSCAN queue in time to meet their deadlines, then we need to allow extra time in the budget for requests current in the L-CSCAN queue to drain to make room for real-time requests. The worst-case time in the L-CSCAN queue is a function of the number of requests in the queue and the *rate* at which those requests are served.

We based L-CSCAN on CSCAN because of its fairness properties [20, 28] and because it interacts well with lower-level schedulers [9] yielding nearly optimal overall system performance. However, we could not use CSCAN because it occassionally has fairness problems and potentially unlimited worst-case performance, particularly when the request stream contains a significant fraction of sequential requests. The behavior that results is that once the CSCAN scheduler reaches the first sequential request, the scheduler keeps servicing the sequential requests. Since each dispatch of a sequential request will often result in the submission of another sequential request, the CSCAN scheduler can be "stuck" servicing the sequential requests for arbitrary and unbounded periods of time. This means that the traditional CSCAN architecture may offer no *guarantees* as to worst-case queueing delay. The solution is to limit the time duration of a single scan, and once that limit is reached, new requests are scheduled for the next scan regardless of their location on the device.

The burstiness or variability of the real-time load impacts the worst-case time budget allocation. One option is to measure the burstiness, and another option is to have the application declare its burstiness. Similar to the L-CSCAN queue, the time a real-time request spends in the SCAN-EDF queue is a function of the length of the SCAN-EDF queue and the worst-case throughput rate. The purpose of the SCAN-EDF queue is to absorb bursts, and hold the real-time requests until they may be submitted to the L-CSCAN queue. Using a SCAN-EDF queue ensures that real-time requests are submitted to the L-CSCAN queue according to deadline and reduces the probability that real-time requests that are submitted to CSched out of order will miss their deadlines. In addition, when the request stream is very bursty, so many requests share a common deadline, requests with the same deadline are sorted in SCAN order, which makes it easier for the L-CSCAN queue to process the real-time requests in a single pass, or at least in as few passes as possible.

5 CSCAN

CSCAN is a well known, popular, and efficient disk scheduling algorithm. For random request streams, it has excellent fairness characteristics, and optimal or near optimal performance, especially when used in a layered architecture with a positional-aware scheduling algorithm at the device[9].

CSCAN consists of two algorithms: *Add* and *Pop*, as shown in Figures 8 and 9, which add a new request to the queue and schedule the next request respectively.

CSCAN is often implemented using a pair of ordered sets, one current and one next. The current set contains those requests whose addresses are greater than the current disk location, while the next set contains those requests whose disk locations are less than the current disk location. There is also the notion of the current disk head location, `offset`, which sweeps from one end of the disk to the other, before jumping back to the beginning.

```
1: Algorithm Add(Request r)
2: if r.offset < offset then
3:    next.insert(r)
4: else
5:    curr.insert(r)
6: end if
```

Figure 8: Adding a request to CSCAN

Algorithm Add in Figure 8 shows the algorithm for adding a new request to a CSCAN queue. Essentially, if the request's address is smaller than the `offset`, then the request will need to be serviced in the next scan so it is added to the `next` set. Otherwise the request's address is greater or equal to the current head location and the request can still be processed in this scan, so it is added to the current set `curr`.

In our implementation, the current and next sets are implemented using C++ STL set classes which are imnplemented as binary trees, so the `insert`, `delete`, and `smallest` operations each take $O(\log n)$ time. In this case, Algorithm Add in Figure 8 takes $O(\log n)$ time.

```
1: Algorithm Pop()
2: if curr.empty() then
3:    Swap(curr, next)
4: end if
5: result ← curr.smallest()
6: curr.delete(result)
7: offset ← result.offset
8: return result
```

Figure 9: Popping a request from CSCAN

Algorithm Pop in Figure 9 shows the algorithm for choosing the next request to be submitted to the device. First, it checks to see if the current scan is done; and if it is, then it swaps the current and next sets to begin the next scan. It then chooses the smallest request in the current scan and does some basic bookkeeping operations: removing the chosen request from the current set and updating the current head location. Algorithm Pop also takes $O(\log n)$ time. Note that it is possible to create an implementation of the set data structure that merges the two operations `curr.smallest()` and `curr.delete()` into a single operation, or at least that does not require a second traversal of the tree structure, yielding a factor of two performance improvement.

With a uniform random access stream, the worst-case delay in a CSCAN queue is proportional to roughly $2N$, or twice the queue size. With mostly sequential access streams, the worst-case delay may be effectively unbounded.

6 Limited CSCAN

We introduce a new variant of CSCAN, the Limited CSCAN, or L-CSCAN, algorithm which provides hard guarantees on the worst-case queue delay. The behavior that causes starvation is the potentially unlimited duration of a single scan. Assuming that the size the queue is limited, then we can control the length of a single scan with additional tests before adding new requests to the current scan.

Compared to CSCAN, L-CSCAN needs has one new configuration parameter, ζ, and four new state parameters: `start`, $\mathcal{T}_{curr}$, $\mathcal{T}_{next}$, and `closed`, and it relies on accurate worst-case throughput rate time estimates $\tau(r)$ for each request. ζ defines the maximum time a request may wait in the queue, `start` is the time at which the current scan started, $\mathcal{T}_{curr} = \sum_{r \in curr} \tau(r)$ is the sum of the expected throughput rate times for the requests in the current scan, $\mathcal{T}_{next} = \sum_{r \in next} \tau(r)$ is the sum of the expected throughput rate times for the requests in the next scan, and `closed` is a boolean specifying whether new requests may still be added to the current scan.

The worst-case throughput rate time estimate $\tau(r)$ is derived from the experimental assessment of the storage device from Section 3. It would take into account three characteristics of the request: $size$, $read \mid write$, and $sequential$, where $size$ is the request size in bytes, $read \mid write$ specifies whether the request is a $read$ or a $write$ request, and $sequential$ is a boolean that specifies whether the request is sequential or not. In general, $sequential$ may be determined using a simple filter, but in the general case $sequential$ merely states whether there is a pending request in the queue with the same $read \mid write$ state as the current request that addresses the bytes immediately preceding the start address of the current request.

```
1: Algorithm Add(Request r)
2: if offset ≤ r.offset AND ζ/2 < (now() - start) + T_curr
   + τ(r) then
3:    closed ← true
4: end if
5: if r.offset < offset OR closed then
6:    next.insert(r)
7:    T_next ← T_next + τ(r)
8: else
9:    curr.insert(r)
10:   T_curr ← T_curr + τ(r)
11: end if
```

Figure 10: Adding a request to L-CSCAN

Algorithm Add in Figure 10 shows the Limited CSCAN algorithm. The primary differences betwen Algorithm Add in Figure 8 and Algorithm Add in Figure 10

is the test at Line 2 that checks whether adding this request to the current scan would cause the scan to exceed the allowed time. Since all requests will be serviced by the end of the second scan, the maximum time allowed for a single scan is $\frac{\zeta}{2}$. Because the primary differences between Algorithm Add in Figure 8 and Algorithm Add in Figure 10 are the $O(1)$ tests and sums in Lines 2, 3, 7, and 10, the running time is still $O(\log n)$.

```
1:  Algorithm Pop()
2:  if curr.empty() then
3:     Swap(curr, next)
4:     start ← now()
5:     T_curr ← T_next
6:     T_next ← 0
7:     closed ← false
8:  end if
9:  result ← curr.smallest()
10: curr.delete(result)
11: offset ← result.offset
12: T_curr ← T_curr - τ(r)
13: return result
```

Figure 11: Popping a request from L-CSCAN

The primary differences between Algorithm Pop in Figure 9 and Algorithm Pop in Figure 11 are the bookeeping changes in Lines 3 – 7 to reset the various state variables at the beginning of each scan and in Line 12 to update $\mathcal{T}_{curr}$, so the running time is still $O(\log n)$.

The reason we developed L-SCAN rather than using an existing algorithm such as N-Step SCAN [7] is that disk performance is optimized with longer scans. CSCAN yields scans that are as long as possible, and longer in some cases, and as a result its performance is excellent. N-Step SCAN arbitrarily limits the number of requests that can be bundled into a single scan to N, which can provide tighter guarantees on worst-case queue delay time, but the scans would generally be much smaller than necessary, which may result in much worse performance than necessary.

7 CSched

CSched is designed to provide optimal, or near optimal, performance, with minimal performance penalty for offering real-time deadline guarantees. In addition to an accurate device characterization $\{\delta, \tau(r), N\}$ from Section 3, it requires two workload-related configuration parameters: $\{\sigma, \lambda\}$, with σ defined as the maximal burst size in seconds and λ defined as the minimal initial slack time for real-time requests.

The primary internal configuration parameter is the allocation of the available time budget, λ, between the SCAN-EDF, L-CSCAN, and device modules. Since the device time budget is defined solely by the device worst-case service time, this is δ. The next step is to determine the time budget which must be allocated to the SCAN-EDF queue to absorb bursts, and this is simply the time required to process the largest burst, or σ. Finally, the remaining budget is assigned to the L-CSCAN queue, leaving $\zeta = \lambda - \sigma - \delta$.

```
1:  class CSched
2:     LCSCAN* lcscan
3:     Sched** pending
4:     int nqueues
5:     time_t ζ
6:  CSched(time_t σ, time_t λ, time_t δ)
7:     ζ ← (λ − σ − δ)
8:     lcscan ← new LCSCAN(ζ)
9:     pending ← new Sched*[nqueues + 1]
10:    pending[0] ← new SCAN_EDF
11:    for i in 1:nqueues do
12:       pending[i] ← new CSCAN
13:    end for
```

Figure 12: CSched data structure and initialization

Figure 12 describes the CSched data structures and shows how the various elements are initialized.

```
1:  Algorithm Add(Request r)
2:  if T_LCSCAN + τ(r) < ζ/4 then
3:     {L-CSCAN is not full}
4:     lcscan.Add(r)
5:  else if 0 < r.deadline then
6:     {real-time request}
7:     pending[0].Add(r)
8:  else
9:     {best-effort or background request}
10:    pending[r.priority].Add(r)
11: end if
```

Figure 13: Adding a request to CSched

Algorithm Add in Figure 13 shows how requests are added to the CSched queue. Please note that $\mathcal{T}_{LCSCAN} = \mathcal{T}_{curr} + \mathcal{T}_{next}$, and is the sum of the worst-case throughput rate times for all requests currently held in the lcscan queue. First, if adding the request to the L-CSCAN queue would not cause it to exceed the allowed size (as specified in terms of time according to Algorithms Add in Figure 10 and Algorithm Pop in Figure11), then the request is simply added to the L-CSCAN queue. Otherwise the request is added to the appropriate pending queue, which is SCAN-EDF for real-time requests and CSCAN for best-effort and background requests. Note

that this code is general and may use any other scheduling system for these non-real-time requests.

```
1:  Algorithm Pop()
2:  result ← lcscan.Pop()
3:  for i = 0 to nqueues do
4:     if pending[i] ¬empty then
5:        r ← pending[i].Head()
6:        if T_LCSCAN + τ(r) ≥ ζ/4 then
7:           {L-CSCAN is now full}
8:           return result
9:        end if
10:       r ← pending[i].Pop()
11:       lcscan.Add(r)
12:    end if
13: end for
14: return result
```

Figure 14: Popping a request from CSched

Algorithm Pop in Figure 14 shows what happens when one removes a request from the queue to submit it to the storage device. First the algorithm removes the next request from the L-CSCAN queue in line 2. Since removing a request from the L-CSCAN queue likely means that there is now room to add a request from the pending queues, the system iteratively checks each pending queue, starting with the real-time SCAN-EDF queue, looking for requests. So long as there is room to add requests to the L-CSCAN queue, it will do so. As soon as the L-CSCAN queue is full or the pending queues are empty, it stops. Note that the test on line 6 is the same as that on line 2 in Algorithm Pop in Figure 13 above. Also, line 5 merely returns the next request that will be popped from the queue without actually removing the request. It is roughly equivalent to the line `r ← curr.smallest()` at line 9 in Algorithm Pop in Figure 11 above.

Since peak performance is obtained using concurrency of roughly sixty (60) (see Figure 2), and since increasing concurrency beyond this number only increases the device worst-case latency without improving performance, the system must maintain this maximal concurrency so long as there are any queued requests. Secondly, since performance is improved by having larger CSCAN (or L-CSCAN) queues in front of the device (see Figure 4), we need to maximize the L-CSCAN queue size. However, we must limit this queue size to ensure that the worst-case queue delay does not exceed the budget.

The Limited CSCAN, L-CSCAN, scheduler's role is to: (1) submit requests to the device in an optimal fashion, so that the lower-level location-aware scheduler's can better utilize its request reordering capabilities to maximize throughput, and (2) ensure that worst-case scheduling delay is limited so that real-time guarantees may be provided by the solution as a whole.

7.1 SCAN-EDF

The SCAN-EDF queue absorbs bursts, so we can limit the size of the L-SCAN queue to provide the real-time guarantees. The primary reason for using a SCAN-EDF rather than a FIFO or EDF queue is for potentially improved performance, particularly with bursty workloads, and as some small protection against clients who do not necessarily submit real-time requests in order with respect to deadlines.

For example, our workload tends to be very bursty, with new "batches" of data needed every 130ms, where a batch may contain as many as thousands of small requests which are submitted to the storage device in random order with respect to the disk address. In this case, using SCAN within each batch means that the requests are sent to the L-CSCAN queue in a partially optimized fashion, so the overall throughput may be maximized.

7.2 CSCAN schedulers

The CSCAN schedulers hold non-real-time requests until there is space available in the L-CSCAN queue. Requests are passed from these queues in priority order, so for example background requests would only be passed from the background CSCAN queue to the L-CSCAN queue if and only if both the real-time SCAN-EDF and best effort CSCAN queues were empty.

One should note that it is entirely possible, and potentially even desireable to utilize alternative architectures, such as that of *p*Clock [12] to control the submission of requests from various non-real-time streams to the L-CSCAN queue. Basically, if you look at the architecture figure in the *p*Clock paper, you can replace the device block in the paper with our L-CSCAN and device blocks.

8 Results

We demonstrate that CSched's performance is essentially indistinguishable from that of CSCAN, a high performance *non* real-time disk scheduling algorithm [9], except that CSched provides real-time guarantees and has no missed deadlines while CSCAN provides *no* real-time guarantees. We analyze the system performance under two extremum workload conditions: uniform random real-time request streams with various workloads with either uniform random or sequential best-effort request streams. Overall system performance is minimized when both the real-time and best-effort requests streams are uniform random, while a sequential best-effort request stream provides the most adversarial request stream from

a fairness standpoint. In reality, many best-effort workloads will be some mixture of uniform random and sequential requests, so real performance will likely be somewhere between these two extremes.

Our application's workload can be characterized by the case with uniform random real-time and best-effort request streams. However, we will have a mixture of request sizes, rather than a uniform 4kB size.

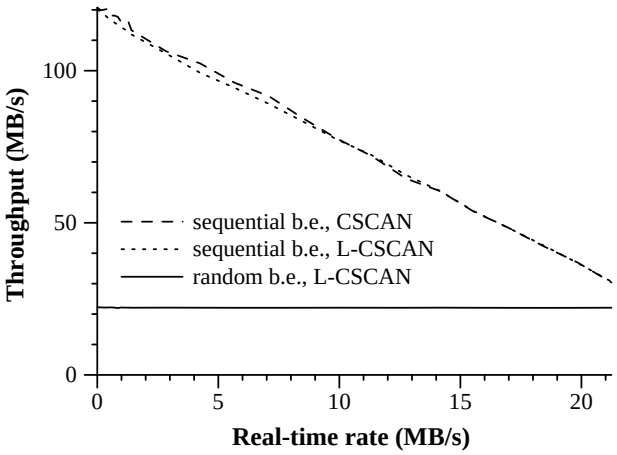

Figure 15: Throughput vs. Real-time rate

Figure 15 shows the throughput for two workloads as a function of real-time data-rate requirements when using a standard CSCAN queue. The first workload is a uniform random request pattern for both the real-time and best-effort requests, and the second workload is a uniform random request pattern for the real-time requests and a sequential pattern for the best-effort requests. The real-time request stream is "smooth", meaning that if the real-time data rate is XMB/s, then the real-time 4KB requests are submitted every $\frac{4096 \cdot 1000}{X}$ seconds. Real-time requests have an initial slack time of thiry (30) seconds, and CSched is configured with zero (0) seconds allocated to SCAN-EDF and 1.4 seconds allocated to the device worst case, leaving 28.6 seconds maximal latency for L-SCAN.

The first thing to notice is that the performance for the uniform random workload is invariant to the real-time request load. The fact that this curve is flat with respect to real-time data rates, and that its value matches the optimal value observed for this workflow (the right-most point of the dotted curve in Figure 4), demonstrates that CSched obtains optimal achievable performance for this workload on this device.

As expected, as the real-time request rate increases, the total system throughput with the sequential best-effort request stream decreases, as the real-time requests absorb an increasing proportion of the available resource. However, the two curves, the dotted and the dashed curves, which represent the optimal performance with and without real-time guarantees respectively, demonstrate that CSched also obtains nearly optimal throughput for this other extreme workload.

As an aside, in the experiments above, L-CSCAN was configured to allow a maximal latency of 28.6 seconds, so at any given time L-CSCAN could hold at most $\frac{28.6}{4}$ seconds worth of requests. On our device with 4KB random requests, the throughput rate time is 0.39ms, which implies that our L-CSCAN queue holds roughly $\frac{28.6}{4 \cdot 0.00039} \approx 18,333$ requests.

For comparison to any or all real-time disk scheduling algorithms that do not permit concurrent requests at the device, please see Figures 3 and 4. The solid line in Figure 3 shows the maximal performance for real-time scheduling algorithms on this device which do not allow concurrent requests at the device across the range of request sizes. The solid line in Figure 4 shows the range of performance for real-time scheduling algorithms which do not allow concurrent requests at the device for 4kB requests, from 0.77MB/s for fully random requests at the left-most point (EDF-like behavior), to 1.37MB/s for efficiently scheduled requests at the right-most point. Conversely, the dotted line in Figure 3 shows the achievable performance range for algorithms that do allow concurrent requests at the device. The dotted line in Figure 4 shows that the range of performance for algorithms that do allow concurrent requests at the device for 4kB requests, is roughly from 9.38MB/s to 19.5MB/s. Note that CSched more than achieves this maximal performance value because of its larger queue size, and the fastest real-time disk scheduling algorithm that does not allow concurrent requests would likely reach 1.4MB/s for the same workload on the same hardware.

Similar observations may be made regarding 1MB request streams from the dashed and dot-dashed curves in Figure 4. EDF would provide about 100MB/s (the left-most point on dashed curve), while the most efficient real-time disk scheduler that did not permit concurrent requests could likely reach about 140MB/s (the right-most point on dashed curve). In comparison, CSched should typically reach about 420MB/s (the right-most point on the dot-dash curve) on the same hardware with the same request stream.

As described above in Section 6, the standard CSCAN scheduler's fairness is impaired as the request stream becomes more sequential and less random. Figure 16 shows the fraction of real-time requests that miss their deadlines when the best-effort request stream is sequential. Clearly, when the fraction of random real-time requests is low compared to the sequential best-effort requests, the random real-time requests get starved. As described earlier, this happens because the best-effort requests may keep entering the CSCAN queue fast enough

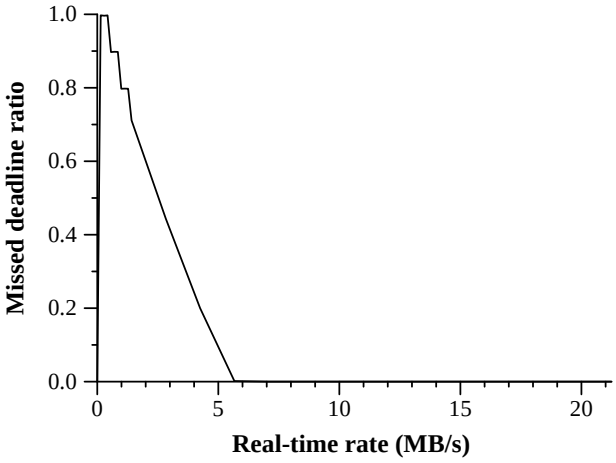

Figure 16: Missed deadlines vs. Real-time rate, Default CSCAN

to cause each scan to spend an inordinate amount of time on the best-effort sequential requests, before it may proceed to service the real-time requests, thereby starving at other addresses. In comparison, L-CSCAN has *no* missed deadlines.

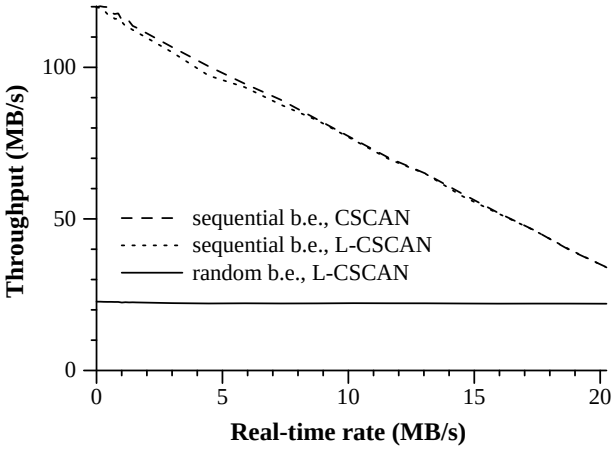

Figure 17: Throughput vs. Real-time rate, bursty real-time request stream

Figure 17 is similar to Figure 15, except that instead of having a "smooth" real-time request stream, it has a bursty request stream, with burst size of five seconds worth of real-time requests at a time. If the worst-case expected throughput rate time for 4KB random real-time requests on our test hardware is 0.39ms ($expected = 0.00039$), and the desired real-time data rate is XMB/s, then number of requests in a given burst is $\sigma \approx \frac{5}{0.00039} \approx 12,820$ requests or 53MB of cache, and the burst inter-arrival rate for 4KB requests is $\frac{4096 \cdot \sigma}{1,000,000 \cdot X}$. If $X =$ 1MB/s, then the burst inter-arrival rate is 52.5 seconds. Clearly, the performance is essentially identical to Figure 15. So, performance is relatively invariant to burstiness in the real-time request stream, so long as the system is properly configured to handle bursts.

One note is in order here. The above calculations are done using the *worst-case* throughput rate time data, which may be substantially different than the *expected* throughput rate time (please see Figure 6). The throughput of the system with the worst-case throughput rate time is about 10.5MB/s, while the expected throughput of the system with thousands of queued requests is roughly 22MB/s.

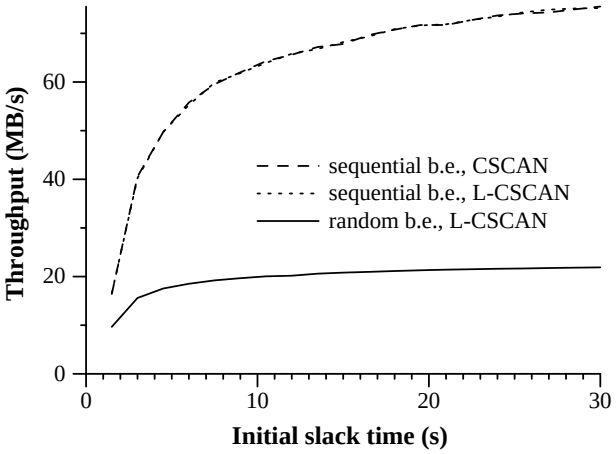

Figure 18: Throughput vs. Initial slack time λ, smooth 9.5MB/s real-time request stream

Figure 18 shows how system throughput varies with λ, with a constant real-time data rate of 9.5MB/s, $\sigma = 0s$, and $\delta = 1.4s$. Since $\lambda \in \{1.5, \ldots, 30\}s$, this means that $\zeta \in \{0.1, \ldots, 28.6\}s$. Clearly, performance improves as ζ and consequently the L-CSCAN queue size increases, which agrees with the results shown in Figure 4 above, namely that increasing the size of the scheduling queue improves the disk performance.

Figure 19 is similar to Figure 15, except that it also shows how the best prior scheduler, CDS-SCAN [37], behaves. Note how CSched and CDS-SCAN's performance is the same when there are no real-time requests, but then how CDS-SCAN's performance on the random best-effort request stream drops while CSched's performance remains constant. Also note how CDS-SCAN's performance for the sequential best-effort stream drops more precipitously than CSched's as the real-time rate increases.

9 Future Work

As λ decreases, a greater fraction of the initial slack time will have to be allocated to the worst-case device service time, leaving less latency allocated to the L-CSCAN

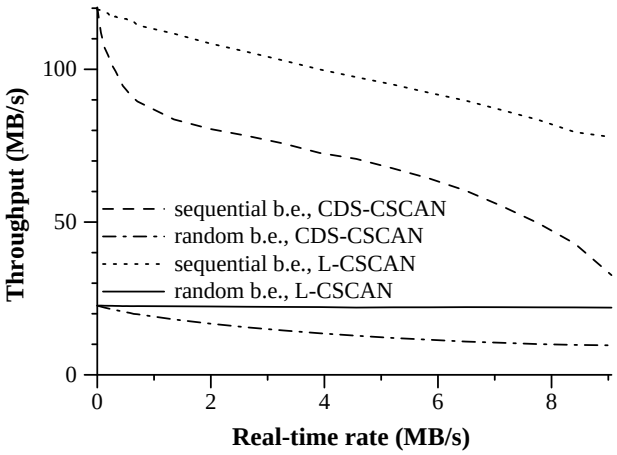

Figure 19: Throughput vs. Real-time rate

queue. This, in turn, implies that the size of the L-CSCAN queue will be reduced, negatively impacting performance. If the real-time workloads are very bursty, i.e. σ is large, then still further time will be "stolen" from the L-CSCAN queue. One avenue for future research is to explore ways to reduce the impact of σ and the average size of the SCAN-EDF queue by allowing real-time requests to pre-emptively remove the most recently added non-real-time requests from the L-CSCAN queue and push them onto a holding stack. Later, as space opens up on the L-CSCAN queue, requests on the holding stack would be re-added to the L-CSCAN queue, thereby mostly preserving fairness among the non-real-time requests.

10 Conclusion

In conclusion, we have presented a new, efficient, $O(\log n)$, real-time disk scheduler that imposes almost no performance penalty over non-real-time optimal schedulers when given sufficient slack time. We demonstrated how the performance may improve as more resources (slack time and buffer space for pending requests) are allocated. We showed how to characterize a device's performance. Given the workload and device characterizations, we have shown how to configure CSched, and demonstrated its performance over a range of workloads. We have also presented L-CSCAN, a modification to CSCAN that preserves fairness with defined maximal queue delay even in the presence of sequential I/O streams, with minimal performance cost.

Acknowledgment

We should like to thank Ron Banner and Mani Fischer for the time and effort they spent reviewing various drafts of this paper, and for their many suggestions which improved the paper. We should also like to thank David Lehavi, Arif Merchant, and Kimberly Keeton for their interesting and stimulating discussions and feedback on this work.

References

[1] SCSI architecture model - 3 (SAM3). Tech. Rep. T10 Project 1561-D, revision 14, International Commitee for Information Technology Standards (INCITS), T10 Technical Committee, Sept. 2004.

[2] CHANG, H., CHANG, R., CHANG, R., AND SHIH, W. Enlarged-Maximum-Scannable-Groups for Real-Time disk scheduling in a multimedia system. In *24th International Computer Software and Applications Conference* (Taipei, Taiwan, Oct. 2000), IEEE Computer Society.

[3] CHANG, H., CHANG, R., SHIH, W., AND CHANG, R. Reschedulable-Group-SCAN scheme for mixed real-time/non-real-time disk scheduling in a multimedia system. *J. Syst. Softw. 59*, 2 (2001), 143–152.

[4] CHANG, R., SHIH, W., AND CHANG, R. Real-time disk scheduling for multimedia applications with deadline-modification-scan scheme. *International Journal of Time-Critical Computing Systems 19* (2000), 149–168.

[5] DEES, B. Native command queuing - advanced performance in desktop storage. *Potentials, IEEE 24*, 4 (2005), 4–7.

[6] DENNING, P. J. Effects of scheduling on file memory operations. In *Proceedings of the April 18-20, 1967, spring joint computer conference* (Atlantic City, New Jersey, 1967), ACM, pp. 9–21.

[7] FRANK, H. Analysis and optimization of disk storage devices for Time-Sharing systems. *J. ACM 16*, 4 (1969), 602–620.

[8] GANG, P., AND CKER CHIUEH, T. Availability and fairness support for storage QoS guarantee. In *Distributed Computing Systems, 2008. ICDCS '08. The 28th International Conference on* (2008), pp. 589–596.

[9] GILL, B. S., AND MODHA, D. S. WOW: wise ordering for writes — combining spatial and temporal locality in non-volatile caches. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies* (San Francisco, California, Dec. 2005), USENIX.

[10] GOPALAN, K. Real-time disk scheduling using deadline sensitive SCAN. Technical Report TR-92, Experimental Computer Systems Labs, Department of Computer Science, State University of New York, Stony Brook, NY, Jan. 2001.

[11] GULATI, A., MERCHANT, A., UYSAL, M., PADALA, P., AND VARMAN, P. Efficient and adaptive proportional share I/O scheduling. *SIGMETRICS Perform. Eval. Rev. 37*, 2 (2009), 79–80.

[12] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: an arrival curve based approach for QoS guarantees in shared storage systems. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (San Diego, California, USA, 2007), ACM, pp. 13–24.

[13] HUANG, Y., AND HUANG, J. Disk scheduling on multimedia storage servers. *Computers, IEEE Transactions on 53*, 1 (2004), 77–82.

[14] HWANG, K., AND CHOI, C. Y. Overlapped disk access for real-time disk I/O. In *Real-Time Computing Systems and Applications, 1999. RTCSA '99. Sixth International Conference on* (1999), pp. 263–269.

[15] INTEL, AND SEAGATE. Serial ATA native command queuing, July 2003.

[16] JACOBSON, D. M., AND WILKES, J. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7, Computer Systems Project, Hewlett-Packard Laboratories, Palo Alto, CA, Mar. 1991.

[17] KELLY, T., COHEN, I., GOLDSZMIDT, M., AND KEETON, K. Inducing models of black-box storage arrays. Technical Report HPL-2004-108, Hewlett-Packard Laboratories, Palo Alto, CA, June 2004.

[18] KESAVAN, M., GAVRILOVSKA, A., AND SCHWAN, K. Differential virtual time (DVT): rethinking I/O service differentiation for virtual machines. In *Proceedings of the 1st ACM symposium on Cloud computing* (Indianapolis, Indiana, USA, 2010), ACM, pp. 27–38.

[19] KIM, T., SONG, E., KOH, Y., WON, Y., AND KOH, K. G-SCAN: a novel real-time disk scheduling using grouping and branch-and-bound strategy. In *Computational Science and Its Applications - ICCSA 2006*, vol. Volume 3980/2006 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, May 2006, pp. 1062–1071.

[20] KUMAR, R. *Fairness in disk scheduling*. Masters thesis, Indian Institute of Science, Bangalore, India, Jan. 1993.

[21] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a Hard-Real-Time environment. *J. ACM 20*, 1 (1973), 46–61.

[22] MERTEN, A. G. *Some quantitative techniques for file organization*. Ph.D. thesis, University of Wisconsin – Madison, 1970.

[23] MESNIER, M. P., WACHS, M., SAMBASIVAN, R. R., ZHENG, A. X., AND GANGER, G. R. Modeling the relative fitness of storage. *SIGMETRICS Perform. Eval. Rev. 35*, 1 (2007), 37–48.

[24] MOLANO, A., JUVVA, K., AND RAJKUMAR, R. Real-time filesystems. guaranteeing timing constraints for disk accesses in RT-Mach. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE* (1997), pp. 155–165.

[25] OZGUR, E., KALLAHALLA, M., AND VARMAN, P. J. Real-time parallel disk scheduling for VBR video servers. In *Proceedings of Fifth International Conference on Computer Science and Informatics* (Feb. 2000).

[26] PAREKH, A., AND GALLAGER, R. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *Networking, IEEE/ACM Transactions on 1*, 3 (1993), 344–357.

[27] POVZNER, A., KALDEWEY, T., BRANDT, S., GOLDING, R., WONG, T. M., AND MALTZAHN, C. Efficient guaranteed disk request scheduling with fahrrad. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008* (Glasgow, Scotland UK, 2008), ACM, pp. 13–25.

[28] PRADHAN, T., AND HARITSA, J. R. Efficient fair disk schedulers. In *Current Trends in Advanced Computing* (Bangalore, India, Dec. 1995), Tata-McGraw-Hill, pp. 236–243.

[29] REDDY, A. L. N., AND WYLLIE, J. Disk scheduling in a multimedia I/O system. ACM, pp. 225–233.

[30] REDDY, A. L. N., WYLLIE, J., AND WIJAYARATNE, K. B. R. Disk scheduling in a multimedia I/O system. *ACM Trans. Multimedia Comput. Commun. Appl. 1*, 1 (2005), 37–59.

[31] RUEMMLER, C., AND WILKES, J. An introduction to disk drive modeling. *Computer 27*, 3 (1994), 17–28.

[32] SARIOWAN, H., CRUZ, R., AND POLYZOS, G. Scheduling for quality of service guarantees via service curves. In *Computer Communications and Networks, 1995. Proceedings., Fourth International Conference on* (1995), pp. 512–520.

[33] SCHINDLER, J., AND GANGER, G. R. Automated disk drive characterization (poster session). In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (Santa Clara, California, United States, 2000), ACM, pp. 112–113.

[34] SEAMAN, P. H., LIND, R. A., AND WILSON, T. L. An analysis of auxiliary-storage activity. *IBM System Journal 5*, 3 (1966), 158—170.

[35] SEELAM, S., AND TELLER, P. Fairness and performance isolation: an analysis of disk scheduling algorithms. In *Cluster Computing, 2006 IEEE International Conference on* (2006), pp. 1–10.

[36] SELTZER, M., CHEN, P., AND OUSTERHOUT, J. Disk scheduling revisited, Jan. 1990.

[37] STAELIN, C., AMIR, G., BEN-OVADIA, D., DAGAN, R., MELAMED, M., AND STAAS, D. Real-time disk scheduling algorithm allowing concurrent I/O requests. Tech. Rep. HPL-2009-244, Hewlett-Packard Laboratories, Oct. 2009.

[38] TSAI, C., CHU, E. T., AND HUANG, T. WRR-SCAN: a rate-based real-time disk-scheduling algorithm. In *Proceedings of the 4th ACM international conference on Embedded software* (Pisa, Italy, 2004), ACM, pp. 86–94.

[39] TSAI, C., HUANG, T., CHU, E., WEI, C., AND TSAI, Y. An efficient Real-Time Disk-Scheduling framework with adaptive quality guarantee. *Computers, IEEE Transactions on 57*, 5 (2008), 634–657.

[40] UYSAL, M., ALVAREZ, G., AND MERCHANT, A. A modular, analytical throughput model for modern disk arrays. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems, 2001. Proceedings. Ninth International Symposium on* (2001), pp. 183–192.

[41] VALENTE, P., AND CHECCONI, F. High throughput disk scheduling with fair bandwidth distribution. *IEEE Transactions on Computers 59* (May 2010), 1172–1186.

[42] VARKI, E., MERCHANT, A., XU, J., AND QIU, X. An integrated performance model of disk arrays. In *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on* (2003), pp. 296–305.

[43] VARKI, E., MERCHANT, A., XU, J., AND QIU, X. Issues and challenges in the performance analysis of real disk arrays. *Parallel and Distributed Systems, IEEE Transactions on 15*, 6 (2004), 559–574.

[44] WANG, M., AU, K., AILAMAKI, A., BROCKWELL, A., FALOUTSOS, C., AND GANGER, G. R. Storage device performance prediction with CART models. In *Proceedings of the joint international conference on Measurement and modeling of computer systems* (New York, NY, USA, 2004), ACM, pp. 412–413.

[45] WORTHINGTON, B. L., GANGER, G. R., AND PATT, Y. N. Scheduling algorithms for modern disk drives. In *Proceedings of the 1994 ACM SIGMETRICS conference on Measurement and modeling of computer systems* (Nashville, Tennessee, United States, 1994), ACM, pp. 241–251.

[46] WORTHINGTON, B. L., GANGER, G. R., PATT, Y. N., AND WILKES, J. On-line extraction of SCSI disk drive parameters. In *Proceedings of the 1995 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems* (Ottawa, Ontario, Canada, 1995), ACM, pp. 146–156.

[47] YU, Y. J., SHIN, D. I., EOM, H., AND YEOM, H. Y. NCQ vs. I/O scheduler: Preventing unexpected misbehaviors. *Trans. Storage 6*, 1 (2010), 1–37.

[48] ZHU, Y. Evaluation of scheduling algorithms for real-time disk I/O. Tech. rep., Department of Computer Science and Engineering, University of Nebraska – Lincoln, May 2002.