Real-time disk scheduling algorithm allowing concurrent I/O requests

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

HP Laboratories
HPL-2009-344

Keyword(s):
real-time disk scheduling, storage systems, operating systems, RAID

Abstract:

We present a real-time disk scheduling algorithm, Concurrent DS-SCAN (CDS-SCAN), which maximizes throughput for modern storage devices by allowing concurrent I/O requests at the device whenever possible. Past real-time disk scheduling algorithms allowed a single request at a time to go to the storage device, which dramatically reduces the utilization and throughput for modern storage devices, such as RAID arrays and disks with efficient positional-aware scheduling algorithms. We extended the DS-SCAN algorithm so that it can properly account for multiple outstanding I/O requests and guarantee real-time constraints for both outstanding and pending real-time requests. We demonstrate CDS-SCAN's performance on a storage array.

External Posting Date: October 21, 2009 [Fulltext] Approved for External Publication
Internal Posting Date: October 21, 2009 [Fulltext]

Real-time disk scheduling algorithm allowing concurrent I/O requests

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

Abstract

We present a real-time disk scheduling algorithm, Concurrent DS-SCAN (CDS-SCAN), which maximizes throughput for modern storage devices by allowing concurrent I/O requests at the device whenever possible. Past real-time disk scheduling algorithms allowed a single request at a time to go to the storage device, which dramatically reduces the utilization and throughput for modern storage devices, such as RAID arrays and disks with efficient positional-aware scheduling algorithms.

We extended the DS-SCAN algorithm so that it can properly account for multiple outstanding I/O requests and guarantee real-time constraints for both outstanding and pending real-time requests. We demonstrate CDS-SCAN's performance on a storage array.

Introduction

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. Over time, systems have become larger and more complex. Firstly, 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. Secondly, 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. Thirdly, disk scheduling may be present in multiple locations in the hardware, such as the disk itself and the RAID controller.

We developed our scheduling algorithm for application-level control of a mix of real-time and best-effort requests on a dedicated storage array. The application controls a manufacturing system with physical objects moving through the system at high speed. Missed deadlines impose a significant cost and have troublesome ramifications on the manufacturing system, so they are unacceptable. During normal operation, the real-time process requires a "batch" of data roughly every 130ms, where a batch may be as little as a single 15MB read I/O, or as many as thousands of generally smaller read I/Os. Usually, we may know as many as several seconds in advance which data will be required for which "batch" at each time. In addition, there are a variety of other processes running concurrently, such as a multitude of writing processes, which write chunks of data of various sizes in a generally bursty fashion, and various bookkeeping and management functions which sporadically read or write data in a bursty fashion. Unfortunately, commodity hardware and operating systems provide no interfaces for specifying either real-time deadlines or even request priorities, so real-time disk scheduling must be implemented in the application.

Initially, engineering attempted to avoid the requirement for a real-time disk scheduler, and to mostly eliminate missed deadlines by radically over-provisioning the storage array and throttling the I/O rates of the non-real-time requests. However, the occurrence of missed deadlines was still unacceptable, and as the manufacturing systems got faster, and the real-time and non-real-time data rate requirements increased, it became clear that only a real-time disk scheduling solution could solve the problem.

Our first implementation used DS-SCAN, which eliminated missed deadlines, but which imposed an unacceptable performance penalty on the overall system throughput compared to the achievable system throughput as found by a simple IOmeter (*iometer.org*) benchmark test. This led to the development of Concurrent DS-SCAN, which largely eliminates that performance penalty.

We found that the storage device may benefit from having multiple outstanding or concurrent requests to work on. If the storage is an array, then often requests will go to different disks, so the storage system may process multiple requests concurrently. However, even for single disks, performance may often be improved by leveraging the in-disk scheduling.

In essence, we are building a multi-level scheduling solution. The top level provides the soft real-time performance guarantees, and submits as many requests as possible (up to a given limit) to the lower-level schedulers in a "scheduler friendly" order. Then the operating system and storage device (or recursively devices, in the case of a storage array), may do their own queuing, trying to optimize throughput for the requests passed to them, oblivious to the real-time aspects and requirements. In addition, by submitting requests to the lower levels in generally circular SCAN order, CDS-SCAN gives the operating system and device-level queuing algorithms a better chance to optimize throughput, since the STF-like scheduling algorithms will be working with sliding "windows" of requests that are clustered spatially as much as possible.

```
rlist rt; /* real-time requests */
rlist be; /* best-effort requests */

request* DSSCAN()
{
  request* r; /* real-time */
  request* b; /* best-effort */

  start_deadlines(rt);

  /* next real-time, CSCAN requests */
  r = EDF(&rt);
  b = CSCAN(&be);

  if (r && r != b
      && r->start_deadline
         < now() + estimate(b))
  {
    /* submit real-time request */
    b = r;
  }
  CSCAN_update(b->offset);

  rtremove(rt, b);
  remove(be, b);

  return b;
}
```

Algorithm 2: DSSCAN queuing algorithm

```
typedef struct rlist {
  struct rlist* next;
  struct rlist* prev;
  struct rlist* rtnext;
  struct rlist* rtprev;
} rlist;

typedef struct request {
  rlist l;
  time_t start_deadline;
  time_t completion_deadline;
  long offset;
  long size;
  void* data;
} request;

void start_deadlines(rlist* rt)
{
  time_t start_deadline = INFINITE;
  for (r = rt->l.rtprev;
       r != rt; r = r->rtprev)
  {
    start_deadline =
      MIN(start_deadline,
          r->completion_deadline)
            - estimate(r);
    r->start_deadline = start_deadline;
  }
}
```

Algorithm 1: Computing start deadlines

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, or elevator algorithm. 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 circular SCAN or CSCAN. In this case, when it reaches the end of the disk, instead of reversing direction it jumps to the beginning of the disk and begins again. This approach has better performance than SCAN and may also result in better fairness.

However, these algorithms overlook one aspect of modern disk drives, namely that rotation delay may be dominant compared to short seek distances. A variety of algorithms, such as shortest time first (STF), grouped shortest time first (GSTF), and weighted shortest time first (WSTF) [3,6], 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 WSTF 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 [12].

There are a wide variety of real-time disk scheduling algorithms. A simple algorithm is earliest deadline first (EDF) [11], where requests are processed according to the deadline order, from earliest to latest. SCAN-EDF [10] 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 or CSCAN order, yielding far higher performance than EDF in this application.

DS-SCAN [1] 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.

A key innovation which enables this hybrid solution is the notion of computing "submission deadlines" from the set of pending real-time requests and their completion deadlines. Since the calculation of the submission deadlines depends upon the entire set of pending real-time requests, it must be recomputed each time a new real-time request is added to the queue, or when a real-time request is removed from the queue and passed on to the device.

More formally, DS-SCAN is described in pseudo-code in Algorithm 1 and Algorithm 2. DS-SCAN uses two linked lists of pending requests: a list of the real-time requests sorted in nearest-deadline order, and a list of all requests (both real-time and best-effort) sorted in ascending disk address order. There is a function *estimate(request)* which estimates the worst-case service time for that request (in ms), and a function *now()* which returns the current time (in ms).

Algorithm 1 demonstrates the computation of the submission deadlines for each real-time request. In actual fact, only the submission deadline for the first request is important as it is used to determine whether the system must submit a real-time request or may submit a request using the CSCAN algorithm.

Algorithm 2 shows the deadline-sensitive scheduling algorithm itself. First, it uses the *start_deadlines()* routine from Algorithm 1 to compute the time by which each real-time request must have been submitted to the storage device in order for them all to complete before their deadlines. There is also some CSCAN logic contained in the routines *CSCAN()* and *CSCAN_udate()*. *CSCAN()* returns the next request according to the CSCAN scheduling algorithm, which is a simple list operation on the best-effort (*be*) list, while *CSCAN_update()* updates the current scan position. Listings for these and other support routines may be found in the Appendix.

The core logic that submits a best-effort request if and only if doing so won't cause a real-time request to miss its deadline is contained in the *if()* statement. This has an implicit assumption that there are no other requests at the device, so that the next scheduling "cycle" will occur no later than *estimate(b)* ms from now.

Note that it is possible to substitute another non real-time scheduling algorithm, such as SSTF, for the relatively simple (but effective) CSCAN algorithm. Also note that it is possible to substitute an alternative real-time scheduling algorithm, such as EDF-SCAN, for the simple EDF scheduler used in DS-SCAN.

Starting in the 1990's, SCSI disk drives started utilizing tagged command queuing (TCQ) [8] capabilities to add support for concurrent requests and internal schedulers with intelligent, positional-aware scheduling algorithms such as STF. Similar capabilities are available in modern SATA drives via NCQ [7] capabilities. This means that it may be more efficient to submit multiple requests to the disk drive and allow it to schedule the requests, rather than to rely solely on the operating system's disk scheduler, since the disk may more easily utilize the low-level detailed information regarding physical location of blocks on disk, rotational position of the platter, location of the seek head, and so forth. While it is possible to build a model of a disk at the higher levels, such as the operating system, and use that model to obtain performance nearly as good as schedulers with access to the true information [13], more complex storage systems such as RAID arrays make such modeling difficult.

Another trend in storage systems is the migration to storage arrays, usually using some form of RAID approach for reliability. The controllers often have significant caches in the controller, and may have significant non-volatile cache for write performance. Storage arrays are often large, shared arrays accessed via a "storage area network" (SAN). Storage is usually managed using logical devices within the array, which can be re-sized as needed to accommodate growth and other requirements. In addition, they often include TCQ functionality, so clients may submit multiple concurrent requests to the same logical device. These large arrays may also have their own scheduling and queuing system as the controller must map the high-level user requests on logical disks into one or more low-level disk requests.

This means that even physical clients of the storage array have at least a three-level scheduling solution: client operating system, controller, and disk. 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 and potentially a second scheduler in the virtual monitor.

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.

```
rlist rt; /* real-time requests */
rlist be; /* best-effort requests */
rlist or; /* outstanding requests */

request* CDSSCAN()
{
  request* r; /* real-time */
  request* b; /* best-effort */
  time_t nod, eo;

  start_deadlines(rt);
  nod = nearest_outstanding_deadline();
  eo = estimate_outstanding();

  /* next real-time, CSCAN requests */
  r = EDF_CSCAN(&rt);
  b = CSCAN(&be);

  if (!b) return NULL;

  if (r && r != b
      && r->start_deadline
         < now() + eo + estimate(b))
  {
    /* submit real-time request */
    b = r;
  }

  if (nod
      && (nod < b->start_deadline
          || !b->start_deadline)
      && nod < now() + eo + estimate(b))
  {
    /* will miss outstanding deadline */
    return NULL;
  }

  CSCAN_update(b->offset);

  rtremove(&rt, b);
  remove(&be, b);

  return b;
}
```

Algorithm 3: CDS-SCAN queuing algorithm

Algorithm

We built Concurrent DS-SCAN (CDS-SCAN) on DS-SCAN, described above, in order to try and allow the system to have multiple requests active at the storage device. This has two major consequences. First, the estimate of when the request will complete must include not only the worst-case service time of the request itself (as embodied in *estimate(req)*), but also the worst-case service time for all outstanding requests already sent to the device. Second, in addition to checking whether sending a new best-effort request to the device might cause the next real-time request to miss its deadline, we must also check all outstanding real-time requests already sent to the device to ensure that none of them will miss their deadlines.

```
int max_out; /* max outstanding */
int num_out; /* num outstanding */
time_t last_started;

void schedule()
{
  request* r = CDSSCAN(); /* request */

  while (r && num_out < max_out)
  {
    append(or, r);
    if (num_out == 0)
      last_started = now();
    num_out++;
    disk_submit(r); /* send to disk! */
    r = CDSSCAN();
  }
}

/* when an I/O completes, call this */
void completion(request* r)
{
  last_started = now();
  num_out--;
  remove(or, r);
  schedule();
}

/* to submit a new I/O to the system */
void submit(request* r)
{
  if (0 < r->deadline)
    EDF_CSCAN_insert(&rt, r);
  CSCAN_insert(&be, r);
  schedule();
}
```

In addition, since our application tends to have periodic deadlines, which may have many requests with the same deadline, we added logic to the real-time queue to add a secondary sort key to the real-time disk request queue on the request disk address (the primary key is the completion deadline). This essentially merges SCAN-EDF functionality for real-time requests, and means that when there are lots of real-time requests with the same deadline, and the system must start processing requests from the real-time queue, then those real-time requests are processed in CSCAN order, thereby preserving some semblance of performance.

From a data structure point of view, we now need to keep track of not only the pending real-time and best-effort requests, but also of all the outstanding requests already sent to the device. We add two new routines, *nearest_outstanding_deadline()*, which returns the nearest completion deadline for outstanding requests, and *estimate_outstanding()*, which returns the expected

worst case completion time for all outstanding requests respectively.

estimate_outstanding() computes the sum of the estimated service times of all outstanding requests. If there are no outstanding requests, then it returns zero. Otherwise, it adjusts this sum for the amount of time the storage device may have already been working on a request, which is the difference between *now()* and the time the most recent request might have been started. This is tracked using *last_started,* which is updated each time a request completes, and each time a request is submitted to the storage device if there are no other outstanding requests.

Pseudo-code for CDS-SCAN is shown in Algorithm 3, and is very similar to DS-SCAN from Algorithm 2 above. The key difference between DS-SCAN and CDS-SCAN is the second *if()* statement, where the algorithm checks to see if submitting a new request to the device could cause an existing real-time request to miss its deadline, in which case it does not submit a request, unless the new request has the same (or earlier) deadline than the outstanding request that is in danger of missing its deadline.

Algorithm 4 shows the external logic keeping track of the outstanding requests, and also submitting as many requests as possible to the storage device, subject to the constraints that real-time requests won't miss their deadlines, and that no more than *max_outstanding* requests are sent to the device at once. Note that this logic is unnecessary for DS-SCAN, because it never has more than a single outstanding request at a time. *completion()* is called each time a disk request completes, and *submit()* is called to submit new requests. *schedule*() sends requests to disk based on the CDSSCAN algorithm, and it is called by both *submit*() and *completion*(), as either event may cause requests to be sent to disk. Note that *schedule()* may submit multiple requests at once. For example, if a real-time request which is near its deadline is at the device, so no other requests can get submitted to the device, and the device happens to service that request last, then the device may have only one outstanding request. As soon as that request completes, assuming there are no other real-time requests in danger of missing their deadlines, then the scheduler should submit as many as *max_outstanding* requests to the device, to maximize throughput.

We may extend CSD-SCAN to allow multiple priorities of best-effort I/O request. This is useful because it allows the solution to ensure that low-priority background processes, such as storage bookkeeping or reorganization processes, do not interfere with normal operation. Instead of having a single SCAN queue, which contains all real-time and best-effort requests, we maintain an array of SCAN queues, one per priority level. Real-time requests are added to the highest priority CSCAN queue. Best-effort requests are added to the CSCAN queue for their priority, with the highest priority requests sharing their queue with the real-time requests. CDSSCAN() in Algorithm 3 is modified so that the single SCAN queue (`be`) becomes an array of queues (`be[i]`), and the best-effort scheduling:

```
b = CSCAN(&be);
```

becomes:

```
for (i = 0, b = NULL; !b && i < N; ++i)
    b = CSCAN(&be[i]);
```

In other words, it looks for the highest priority queue with any requests, and chooses a request from that queue in CSCAN order.

Analysis

First, we show that DS-SCAN satisfies the conditions of a real-time scheduling algorithm, with the assumptions that the worst-case estimates are correct and that the real-time requirements are feasible. Then we build upon that result to show that CDS-SCAN also provides real-time guarantees when the device processes multiple requests concurrently, with the additional assumption that a device will not take more time to process requests concurrently that it would do to process them sequentially.

For real-time disk scheduling algorithms generally, and DS-SCAN and CDS-SCAN in particular, it is necessary that the worst-case estimates be correct, meaning that the request never takes longer than its worst-case estimate. That this condition is necessary is obvious, as if the estimate is too short, then the system may, in good faith and in expectation of meeting its obligations, submit the request to the storage device worst-case estimate time before the deadline, but since processing took longer, the system missed its deadline.

Before formalizing the concept of real-time scheduling feasibility, we first analyze the DS-SCAN algorithm and its notion of submission deadlines. The analyses below assume that there is an initial real-time queue, and that the scheduler must process it according to the deadlines. It also assumes that the scheduler only submits a single request at a time to the disk. Later we will analyze the case of adding new requests to the queue.

Lemma 1: *A disk request submitted to the disk on or before its submission deadline will meet its completion deadline.*

Since the submission deadline is never later than the completion deadline less its worst-case service time, it may never complete after its completion deadline. ■

Lemma 2: *If the submission deadline for the first real-time request is not in the past, then that request will always be submitted to the disk on or before its submission deadline*.

For the first real-time request to be submitted after its submission deadline, another request must have been submitted to disk before that request, and its service time must have been long enough to cause the scheduler to miss its submission deadline. Since the scheduler will only submit a request to disk other than the earliest real-time request if the current time plus the worst-case service time is less than the submission deadline of the earliest real-time request, then this cannot occur. ■

A feasible queue is one for which all real-time requests can meet their deadlines. The easy test for a feasible queue is that the submission deadline for the request with the earliest completion deadline is no earlier than the current time. Conversely, it is easy to see that a real-time queue for which the earliest submission deadline is in the past cannot provide any real-time guarantees and may miss one or more real-time deadlines.

For example, take two requests, one with a deadline 4ms in the future and one with a deadline 5ms in the future, and each with a worst-case service time of 3ms. In this case, the submission deadline of the later request is 2ms in the future, and the submission deadline of the earlier request is 1ms in the past. If we submit the first request to disk immediately, and it takes the worst-case service time, then that request will meet its deadline, completing 3ms in the future. If the later request is then immediately submitted to disk and it too takes the worst-case time to service, then that second request will miss its deadline and complete 6ms from the start, 1ms after its deadline.

However, the worst-case estimate is just that, a worst-case estimate, so it is likely that one or more of the requests will complete more quickly than estimated, and so all requests may complete before their deadlines even though the system cannot provide any real-time guarantees. In the example above, if either request completes in 2ms instead of the worst-case 3ms, then both requests will meet their deadlines.

Theorem 1: *All requests in a real-time request queue for which the submission deadline of the request with the earliest deadline is no earlier than the current time will complete on or before their deadlines*.

If there is still time to submit the earliest request to disk on or before its submission deadline, then according to *Lemma 2* above that request will be submitted to disk on or before its submission deadline, then from *Lemma 1* it will therefore complete on or before its completion deadline. Since the submission deadline of each subsequent request may be no earlier than the completion time of the previous request, at the completion of each request, the current time will be no later than the submission deadline of the next real-time request, and so by *Lemma 2* each real-time request will be submitted to disk on or before its submission deadline and it too shall complete on or before its completion deadline according to *Lemma 1*. ■

If we look at a dynamic system, where new requests get added to the queue over time, then each time a new real-time request is added to the queue, the submission deadlines are re-calculated. So long as the nearest submission deadline is not in the past, the queue remains feasible, and the scheduler may give real-time guarantees for that queue.

Now we come to the CDS-SCAN analysis, where multiple requests may be sent to the device concurrently. Since there are no priority or deadline mechanisms in the SCSI or ATA protocols, and since the devices have scheduling algorithms internally, we assume that the device may process concurrent requests in any order, either sequentially, or concurrently.

We also add a second condition, namely that the storage device never takes longer to process concurrent requests than it would take to process the requests in sequence. In practice this is true, and for a device to behave otherwise would be considered a bug. This allows us to compute the worst-case service time for a set of concurrent requests as the sum of the individual worst-case estimate. This condition may be shown to be necessary for real-time scheduling by counter-example. Assume that there is a real-time request at the device, whose deadline is just beyond the sum of the worst-case estimates for the real-time request already being serviced and a candidate best-effort request under consideration for submission. CDS-SCAN will submit the second best-effort request to the device since the sum of the worst case estimates is less than the nearest deadline. If the device takes longer to service the two requests than it would have done were the requests submitted in sequence, then the real-time request will miss its deadline.

For the purposes of determining the worst-case completion time for real-time requests when there are multiple concurrent requests, we must assume that the device will service the requests sequentially and that it will service any real-time requests last. This means that the worst-case expected completion time for real-time requests at the device is the sum of the worst-case service times for all requests at the disk.

Unlike the DS-SCAN, we must keep track of how much time a device may have spent servicing outstanding

requests. In the DS-SCAN case the device is always idle when a request is passed to it from the scheduler. In the CDS-SCAN case, the device may have already been working on another request, so the current worst-case service time estimate must be updated to account for time already spent servicing the request. Again, for the purposes of worst-case analysis, since we assume that the device services requests sequentially, the time spent processing outstanding concurrent requests is the shorter of the elapsed time since the device last completed a request, and the elapsed time since the device was idle.

We define the *outstanding service time* as the sum of the worst-case service times for all requests at the disk less the elapsed service time for the outstanding requests. This is the worst-case estimate for the remaining time until the device finishes servicing the currently outstanding requests, assuming that no other requests are sent to the device.

We will analyze the system assuming that no new requests are added to the system, and later we will extend the analysis to the dynamic case where new requests are added to the queues and devices.

Lemma 3: *A real-time request which has been submitted to disk will complete on or before its completion deadline so long as its completion deadline is no earlier than the outstanding service time*.

Assume otherwise. Then there is a real-time request which completed after its deadline. Since the deadline is later than the sum of the worst-case estimates for all requests in the device, less the elapsed service time, then either one or more requests took longer than their worst-case estimate, or the device took longer to service concurrent requests than it would have done to service them sequentially. ■

Lemma 4: *If the submission deadline for the first real-time request is no earlier than the outstanding service time, then that real-time request will always be completed on or before its completion deadline*.

Assume otherwise. Either the request which missed its completion deadline was submitted to the device after its submission deadline, or it was submitted on or before its submission deadline and only completed after its completion deadline.

For the request to have been submitted after its submission deadline, then either the device took longer than the outstanding service time to complete the servicing of the outstanding requests, or the scheduler submitted another request before the first real-time request. However, the scheduler cannot have submitted another request before the real-time request because CDS-SCAN will only allow another request if the outstanding service time plus the worst-case estimate of the additional request is less than the submission deadline of the nearest real-time request.

If the real-time request was submitted on or before its submission deadline, then by *Lemma 3* it must complete before its completion deadline so long as no other requests are added to the device. But, no requests may be added to the device unless the outstanding service time plus the worst-case estimate of the new request are less than the completion deadline of the real-time request. ■

Theorem 2: *Starting from a state where all outstanding real-time requests already submitted to the storage device have completion times no later than the outstanding service time, all requests in a real-time request queue for which the submission deadline of the request with the earliest deadline is no earlier than the outstanding service time will complete on or before their deadlines*.

From *Lemma 3* we can show that all outstanding real-time requests will be completed on or before their completion deadlines, since the scheduler will not send any requests to the device if it would cause the outstanding service time to be later than the earliest completion deadline of the outstanding requests.

Similarly, we can show that since the outstanding service time is no greater than the submission deadline of the nearest real-time request, then the request will be submitted to the device on or before its submission deadline because of the CDS-SCAN constraint that other requests may only be submitted if and only if the outstanding service time plus the worst-case estimate is no later than the submission time of the earliest real-time request. Therefore the nearest real-time request will be sent to the disk on or before its submission deadline, and from *Lemma 4* it can be seen that the request will meet its deadline. ■

Similar to the DS-SCAN case, so long as the nearest submission deadline remains no earlier than the outstanding service time, the schedule will be feasible and the scheduler will be able to provide its real-time guarantees. Once that condition is violated, the system may no longer guarantee real-time performance. However, once the system reaches a state where the nearest completion deadline for outstanding requests is no later than the outstanding service time, and the nearest submission deadline is no earlier than the outstanding service time, then it will again be able to provide real-time guarantees.

Results

The following experiments were conducted on a HP DL360 server with two dual-core Intel processors and 4GB of RAM running Windows Server 2003. The

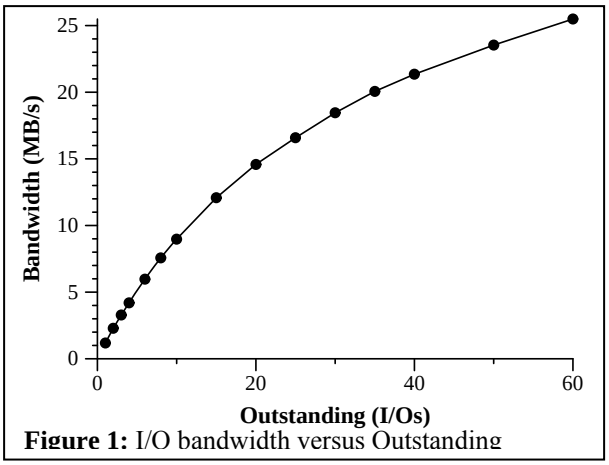

Figure 1: I/O bandwidth versus Outstanding

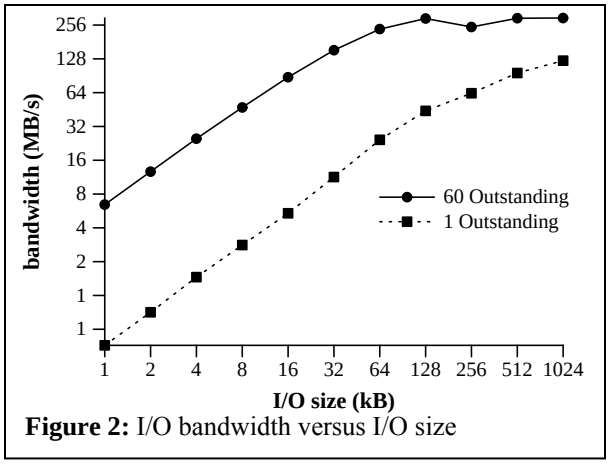

Figure 2: I/O bandwidth versus I/O size

storage array used for the performance testing was built from twelve 146GB 10k RPM SAS disks, attached on two SCSI cables (ten disks on one cable and two on the other) to an HP P600 storage controller. The storage was configured into a RAID0 array with 128kB stripes and a total usable size of 1.6TB. The software accessed the raw device directly, rather than using a file system, and it utilized Windows' asynchronous I/O capabilities to send multiple I/Os to the device at once.

First, we demonstrate that utilizing available concurrency and in-device scheduling at the storage array and disk levels can yield improved performance. Figure 1 shows I/O bandwidth as a function of the available I/O concurrency for 4kB random disk reads. In this experiment, there is no software-level queuing or scheduling. There is a closed queuing model with the same number of requests as the desired concurrency level, so as soon as one request completes, the next request is submitted directly to the storage device. The first curve has no software-level queue and keeps the number of outstanding I/Os at the device constant by submitting a new I/O request as soon as one completes.

Clearly there is a significant performance benefit to concurrent I/O at the device level, with a greater than twenty-fold improvement in I/O bandwidth between no concurrency (a single outstanding request at a time), and full concurrency (in this case, sixty outstanding requests). In addition, it seems clear that maximally utilizing the available concurrency is the best strategy for overall throughput, at least for this device.

Figure 2 shows throughput as a function of I/O size for no concurrency and maximal concurrency. Clearly, throughput is better across a broad range of I/O sizes, with the gap only beginning to narrow once the system reaches a bottleneck at about 250MB/s.

Next, we need to verify that the real-time scheduler may utilize this available concurrency with minimal cost and no unnecessary missed deadlines. (If the application requests data faster than the device can deliver, then you will have missed deadlines, but they are not the fault of the scheduler and would be called necessary missed deadlines.)

To demonstrate this, we use a closed queuing system (a new request is submitted as soon as a previous one is completed) of best-effort request requests of given size and uniform random placement. We also include a stream of real-time requests of a certain bandwidth. Figure 3 shows the overall (best effort plus real-time) throughput and the real-time throughput as a function of the requested real-time throughput for 4kB requests, with a set of five hundred best effort requests, and a maximal concurrency of sixty (60). The best-effort throughput is the difference between the real-time and overall curves. During all of our tests, the scheduler never missed a real-time deadline.

Clearly, the system is able to provide the desired real-time data rate, almost up to the entire available bandwidth of the system. There is some degradation of overall system throughput as the real-time data rate starts becoming a significant fraction of the overall system throughput. Remember from Figure 1 that if we did not use the available hardware concurrency, the maximal system throughput for this configuration with 4kB read requests would be about 1MB/s, which is the starting point for the requested real-time data rate in Figure 3.

A different way of explaining the performance dip centered around a real-time data rate of about 12MB/s is by looking at the level of concurrency that the scheduler was able to achieve at various real-time data rates. Figure 4 shows the cumulative density function of the concurrency for the experimental results from Figure 3, at 1MB/s, 12MB/s, and 22MB/s respectively.

At 1MB/s, the scheduler is clearly able to process the real-time requests in a timely fashion, and is able to fully utilize the allowed concurrency and therefore to obtain optimal throughput. By 12MB/s, slightly less than half the maximal throughput, the scheduler has to

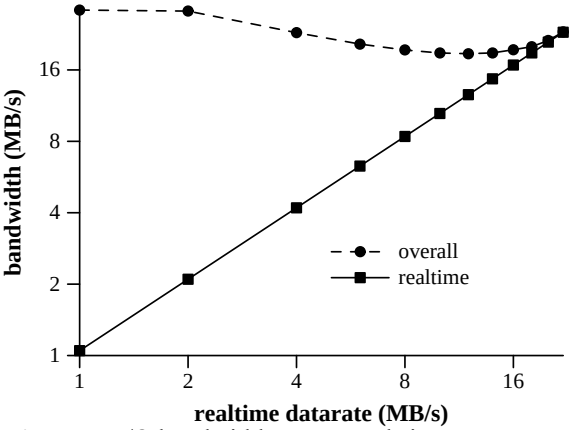

Figure 3: I/O bandwidth versus real-time

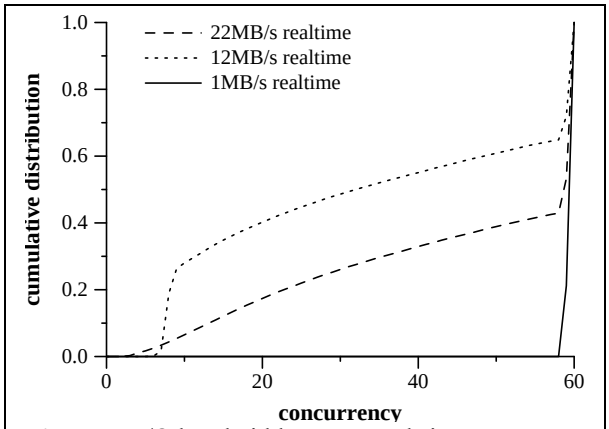

Figure 4: I/O bandwidth versus real-time

block best-effort requests while real-time requests that are in danger of missing their deadlines finish on the device. Looking at the dotted line, one may clearly see that the device spends about forty percent of its time with a concurrency level of less than 20. Again, referring back to Figure 1, it is clear that this loss of concurrency due to the real-time scheduling constraints penalizes overall system throughput.

Interestingly, as the real-time rate increases further, the system is able to again increase the level of concurrency in the system. This is in part because most of the requests are real-time requests, and more of them are in danger of missing deadlines, so the scheduler may submit them to the device even though other real-time requests are already at the device.

Acknowledgements

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.

Conclusions and Future Work

We have presented the CDS-SCAN algorithm, a soft real-time disk scheduling algorithm for mixed real-time and best-effort workloads, and we have shown that CDS-SCAN generally increases concurrency and may dramatically improve performance over DS-SCAN.

Future research might focus on developing an optimal multi-level scheduling policy. This is especially difficult when a mix of storage arrays, solid state disks, and variations in intra-disk scheduling algorithms is taken into account.

Other future work might include developing an experiment-based model of worst-case performance for complex storage systems. Essentially, this would extend models such as Disk Mimic [13] to more complex storage devices. However, for the purposes of submission deadline computation, only worst-case estimation is necessary.

References

[1] Kartik Gopalan and Tzi-cker Chiueh. *Real-time disk scheduling using deadline sensitive SCAN*. Technical Report TR-92, Experimental Computer Systems Labs, Dept. of Computer Science, State University of New York, Stony Brook, NY, January 2001. (available online as http://www.ecsl.cs.sunysb.edu/tr/TR92.ps.gz).

[2] Lars Reuther and Martin Pohlack. *Using SATF in real-time systems*. Work-in-Progress Report, 2nd USENIX Conference on File and Storage Technologies (FAST 03), (San Francisco, CA, March 31–April 2 2003). March 2003. (available online as http://os.inf.tu-dresden.de/~mp26/publications/fast2003.pdf)

[3] D. M. Jacobson and J. Wilkes. *Disk scheduling algorithms based on rotational position*, Technical Report HPL-CSP-91-7, HP Labs, 1991. (available online as http://www.hpl.hp.com/research/ssp/papers/HPL-CSP-91-7rev1.pdf)

[4] Alexander Thomasian and Chang Liu. Disk scheduling policies with lookahead. *ACM SIGMETRICS Performance Evaluation Review*, (30)2, pages 31-40. September 2002.

[5] Saman Zarandioon and Alexander Thomasian. Optimization of online disk scheduling algorithms. *ACM SIGMETRICS Performance Evaluation Review*, (33)4, pages 42-46. March 2006.

[6] Margo Seltzer, Peter Chen, and John Ousterhout. *Disk scheduling revisited*. Proceedings of the Winter 1990 USENIX Technical Conference (Washington, DC, 22–26 January 1990), pages 313–323.

[7] Amber Huffman and Joni Clark. *Serial ATA native command queuing*. Seagate and Intel joint white paper. July 2003. (available online as http://www.seagate.com/content/docs/pdf/whitepap

er/D2c_tech_paper_intc-stx_sata_ncq.pdf)
[8] SCSI Architecture Model – 3 (SAM3). T10 Project 1561-D, revision 14. International Committee for Information Technology Standards (INCITS), T10 Technical Committee. Reference ISO/IEC 14776-413-200X. September 2004.
[9] Kitae Hwang and Chang Yeol Choi. *Overlapped Disk Access for Real-time Disk I/O*. Proceedings of the Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99), 1999. pp. 263-269.
[10] A.L. Narasimha Reddy and Jim Wyllie. *Disk scheduling in a multimedia I/O system*. Proceedings of the first ACM International Conference on Multimedia (Anaheim, CA), 1993. pages 225-233.
[11] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, pages 46-61, 1973.
[12] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. *Scheduling algorithms for modern disk drives*. Proceedings of the ACM SIGMETRICS Conference. May 1994. pp. 241-251.
[13] Florentina I. Popovici, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. *Robust, portable I/O scheduling with the disk mimic*. Proceedings of the USENIX 2003 Annual Technical Conference (San Antonio, TX, 9-14 June 2003).

Appendix

Below are listings for various support functions used in Algorithms 1-4 above. Note that a real-time request will belong to two lists, hence the pair of <next,prev> and <rtnext,rtprev> list pointers in the request structure.

```
long CSCAN_offset;

/* initialize a circular linked list */
void rlist_initialize(rlist* l)
{
    l->next = l->prev = l;
    l->rtnext = l->rtprev = l;
}

/* inserts r after l */
void insert(request* l, request* r)
{
    r->next = l->next;
    r->prev = l;
    r->next->prev = r;
    l->next = r;
}

void rtinsert(request* l, request* r)
{
    r->rtnext = l->rtnext;
    r->rtprev = l;
    r->rtnext->rtprev = r;
    l->rtnext = r;
}

void remove(request* r)
{
    r->prev->next = r->next;
    r->next->prev = r->prev;
}

void rtremove(request* r)
{
    r->rtprev->rtnext = r->rtnext;
    r->rtnext->rtprev = r->rtprev;
}

/* earliest deadline first */
request* EDF(rlist* rt)
{
    if (rt->rtnext != rt)
       return rt->rtnext;
    return NULL;
}

/* insert a new request into EDF queue */
void EDF_insert(rlist* rt, request* r)
{
    request* p = rt->rtnext;
    for (; p != rt; p = p->rtnext) {
        if (r->completion_deadline <
            p->completion_deadline)
        {
            rtinsert(p->rtprev, r);
            break;
        }
    }
    if (p == rt) rtinsert(rt->rtprev, r);
}

/* choose next CSCAN request */
request* CSCAN(rlist* l)
{
    request* r = (request*)l->next;

    if (l->next == l) return NULL;

    for (; (rlist*)r != l; r = r->l.next)
        if (CSCAN_offset < r->offset)
            return r;

    /* CSCAN: jump back to beginning */
    return (request*)l->next;
}

/* add a new request into CSCAN queue */
void CSCAN_insert(rlist* l, request* r)
{
    request* p = (request*)l->next;

    for (; p != l; p = p->l.next)
        if (r->offset < p->offset) {
            insert(p->l.prev, r);
```

```
            break;
        }

    if (p == l) insert(p->l.prev, r);
}

void CSCAN_update(long offset)
{
    CSCAN_offset = offset;
}

/* choose next CSCAN request */
request* EDF_CSCAN(rlist* l)
{
    request* r = (request*)l->rtnext;
    request* d = r;

    if (l->rtnext == l) return NULL;

    for (; r != l; r = r->l.rtnext) {
        if (d->deadline < r->deadline)
            return d;
        if (CSCAN_offset < r->offset)
            return r;
    }

    /* CSCAN: jump back to beginning */
    return d;
}

/*
 * insert a new request into EDF CSCAN
 * queue
 */
void
EDF_CSCAN_insert(rlist* l, request* r)
{
    request* p = (request*)l->rtnext;

    for (; p != l; p = p->l.rtnext)
        if (r->deadline < p->deadline
            || (r->deadline ==
                    p->deadline
               && r->offset < p->offset))
        {
            rtinsert(p->l.rtprev, r);
            break;
        }

    if (p == l) rtinsert(p->l.rtprev, r);
}

/*
 * estimate worst-case processing of
 * requests already sent to device,
 * taking into account time already
 * spent servicing those requests
 */
time_t estimate_outstanding()
{
    request* r = or.next;
    time_t est = 0;
    if (num_out == 0) return 0;

    for (; r != &or; r = r->next)
        est += estimate(r);
    est -= now() - last_started;
    if (est < 0) return 0;
    return est;
}
```

One possible optimization of the system is to realize that it is not necessary to re-compute all of the start deadlines in each scheduling cycle. Instead, it is sufficient to update the start deadline computations for affected real time requests when a new request is added to the real time queue, and when a real-time request is processed before its submit deadline from the best effort queue. As soon as an earlier request's completion deadline is before the start deadline of the current request, one may stop updating the start deadlines. In this case, one would remove the calls to *start_deadlines*() from *CDSSCAN*() and add a call to *update_start_deadlines*() to *submit*().

```
void
update_start_deadlines(
    rlist* rt, request* r)
{
  time_t start_deadline;

  if (r->rtnext != rt) r = r->rtnext;

  start_deadline =
    r->completion_deadline;

  for (; r != rt; r = r->rtprev)
  {
    time_t e = estimate(r);
    if (r->completion_deadline
        <= start_deadline - e)
      break;
    start_deadline =
      r->completion_deadline - e;
    r->start_deadline = start_deadline;
  }
}
```