Efficient I/O Scheduling with Accurately Estimated Disk Drive Latencies

Vasily Tarasov[1], Gyumin Sim[1], Anna Povzner[2], and Erez Zadok[1]
[1]Stony Brook University, [2]IBM Research—Almaden

***Abstract*—**
Modern storage systems need to concurrently support applications with different performance requirements ranging from real-time to best-effort. An important aspect of managing performance in such systems is managing disk I/O with the goals of meeting timeliness guarantees of I/O requests and achieving high overall disk efficiency. However, achieving both of these goals simultaneously is hard for two reasons. First, the need to meet deadlines imposes limits on how much I/O requests can be reordered; more pessimistic I/O latency assumptions limit reordering even further. Predicting I/O latencies is a complex task and real-time schedulers often resort to assuming worst-case latencies or using statistical distributions. Second, it is more efficient to keep large internal disk queues, but hardware queueing is usually disabled or limited in real-time systems to tightly bound the worst-case I/O latencies.

This paper presents a real-time disk I/O scheduler that uses an underlying disk latency map to improve both request reordering for efficiency and I/O latency estimations for deadline scheduling. We show that more accurate estimation of disk I/O latencies allows our scheduler to provide reordering of requests with efficiency better than traditional LBN-based approaches; this eliminates the need of keeping large internal disk queues. We also show that our scheduler can enforce I/O request deadlines while maintaining high disk performance.

I. Introduction

Modern general-purpose computers and large-scale enterprise storage systems need to support a range of applications with different performance and timeliness requirements. For example, audio and video streams in multimedia applications require timely data delivery guarantees, while concurrent interactive applications remain responsive. In large-scale enterprise storage systems, the rise of storage consolidation and virtualization technologies [1], [2] requires the system to support multiple applications and users while meeting their performance constraints. For example, Internet-based services that share a common infrastructure expect I/O performance for each service in accordance with its service level agreement [3].

Managing disk I/O is an essential aspect of managing storage system performance, as disks remain a primary storage component and one of the top latency bottlenecks. A classic way to improve disk performance is to reorder disk I/O requests, because disk performance largely depends on the order of requests sent to the disk device. With an additional requirement of providing timeliness guarantees, the traditional goal of maximizing overall disk efficiency remains an important requirement. As a result, many real-time disk I/O schedulers [4], [5], [6] combine reordering algorithms (such as SCAN) with real-time scheduling (such as EDF) to optimize disk performance while meeting guarantees. Similarly, fair- or proportional-sharing schedulers reorder some requests to improve disk efficiency.

Since operating systems have a limited knowledge of disks, existing disk I/O schedulers perform reordering based on the requests' Logical Block Number (LBN). I/O schedulers assume that the larger the difference between two LBN addresses, the longer it takes to access the second LBN address after accessing the first one. Although this assumption used to be reasonable in the past, we will demonstrate that it no longer holds and is misleading due to complex specifics of the modern disk drive design. The disk drive itself has an internal queue and a built-in scheduler that can exploit the detailed information about the drive's current state and characteristics. Consequently, built-in disk drive schedulers are capable of performing request scheduling with a higher efficiency than LBN-based schedulers can at the OS level.

Best-effort disk I/O schedulers improve their performance and overcome inefficiencies of LBN-based scheduling by keeping as many requests as possible outstanding at the underlying disk device so they are scheduled by the drive's internal scheduler. However, disk I/O schedulers with real-time guarantees cannot take advantage of the drive's internal scheduler, because the I/O scheduler loses control over requests sent to the drive and the drive's internal scheduler is not aware of the host-level request deadlines. Thus, existing real-time schedulers keep inefficient internal disk queues of only one or two requests, or allow more outstanding requests at the disk drive for soft guarantees, but they require frequent draining of internal disk queues in order to meet request deadlines [7].

If OS would have a more accurate source of information about disk drive latencies, it could perform efficient scheduling while meeting request deadlines. But how large can potential benefits of accurate latency estimation be? In this paper, we first propose a novel request reordering algorithm based on maintaining a disk drive latency map within the OS kernel. The map allows us to accurately estimate the *actual* latency between any pair of LBN addresses anywhere on the disk. We designed and implemented a real-time disk I/O scheduler that meets request deadlines as long as the disk can sustain the required throughput. The scheduler learns the disk latencies and adapts to them dynamically; it uses our request reordering algorithm to maintain high efficiency while meeting request deadlines. Real-time schedulers that use a distribution of I/O execution times over all disk-block addresses, tend to overestimate I/O execution times; in contrast, our disk drive

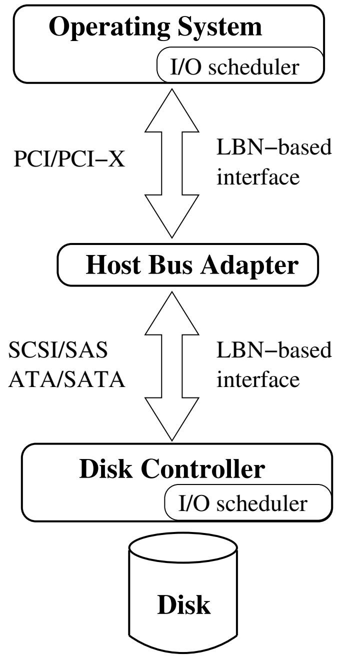


Fig. 1. I/O architecture of commodity servers. The OS accesses the block devices through a standard LBN-based interface that hides the device's physical characteristics.

latency map provides more accurate per-LBN-pair execution time.

We show that while allowing only one request to the underlying disk drive, our reordering algorithm can achieve performance up to 28% better compared to LBN-based schedulers. We also demonstrate that our scheduler enforces request deadlines while providing higher throughput than LBN-based schedulers. We address CPU trade-offs using approximation algorithms and user-defined memory limits. For large datasets our map size can become too large to fit in RAM. Given the benefits of accurate latency prediction as demonstrated in this paper, we expect that various techniques can be used in the future to reduce map sizes and thus provide similar benefits for larger devices.

The rest of the paper is organized as follows. Section II presents experimental results that motivated the creation of our scheduler. In Section III, we describe how latency estimation in our design allows to increase disk throughput for batch applications and enforce deadlines for real-time applications. Section IV details the implementation and Section V evaluates our scheduler against others. In Section VI, we survey related work. We conclude in Section VII and discuss future directions in Section VIII.

II. Background

In this section we describe the overall Input/Output mechanism relevant to the I/O scheduler, deficiencies of this mechanism, and the experiments that led us to create a new scheduler. To the OS I/O scheduler, the disk device appears as a linear array where the Logical Block Number (LBN) is the index

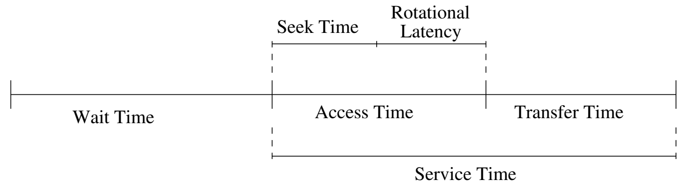


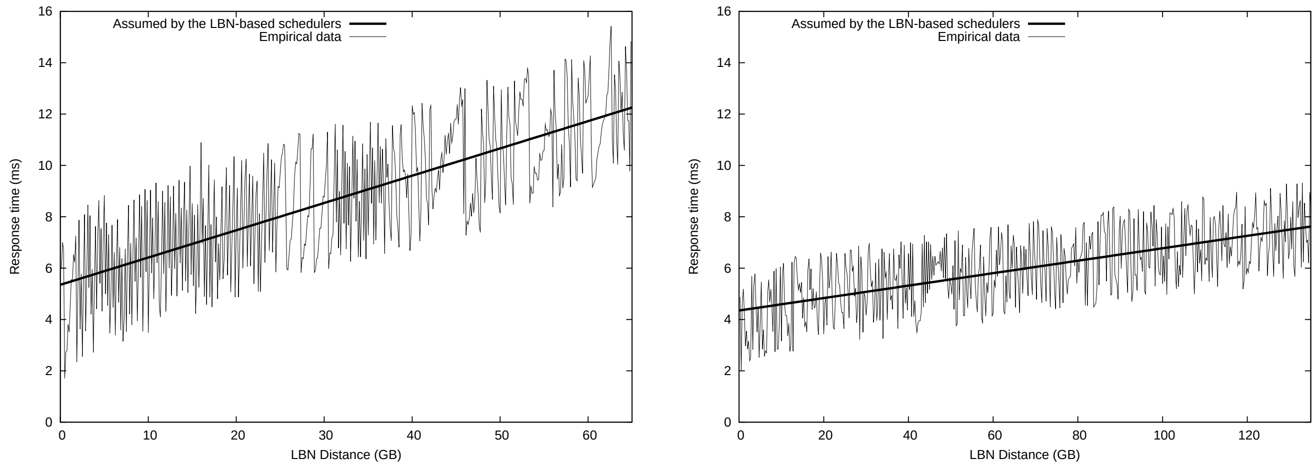
Fig. 2. Decomposition of the request response time.

into this array. Such address representations are used by many I/O protocols (e.g., SATA and SCSI). When the disk scheduler sends a request to the underlying disk device, the Host Bus Adapter (HBA) passes these requests to the disk controller, which in turn maps LBNs to the physical location on the disk. The disk drive has its own internal queue and a scheduler that services requests one by one and then returns completion notifications back to the OS, as seen in Figure 1.

Figure 2 depicts a typical timeline for the request's execution. Response time is the time from the request submission to the I/O scheduler to the request's completion. Response time consists of wait time and service time. Wait time is the time spent in the I/O scheduler's queue. Service time is the time from when the request is picked from the queue for service and until the moment the request completes. Service time consists of access time and transfer time. Access time is required to locate the data; transfer time is the time to transfer the data. For disk drives, the access time consists of the time to position the arm (seek time) and the time until the platter reaches the required position (rotational latency).

Request reordering at the I/O scheduler directly affects access and service times. Indirectly it also affects wait time because shorter service times lead to shorter queues. The OS I/O scheduler knows only about the requests' LBNs and sizes; it is the only criterion OS schedulers can use to perform request scheduling (apart from the knowledge about the request owners, the processes). A common assumption is that the shorter the distance between two LBN addresses is, the smaller is the access time between them. Given this assumption, the scheduler can, for example, sort all requests by their LBNs and submit them in that order (enforcing any required deadlines if necessary).

This assumption used to be true in the early days of disk drives when seek time dominated the rotational latency. Since then, manufacturers significantly improved their disk positioning mechanisms and nowadays rotational latency is of the same magnitude as seek time (e.g., 4msec vs. 2msec for a typical 15K RPM drive). Moreover, the variety of devices and their complexity increased dramatically. ZCAV/ZBR technology, error correction, block remapping, improvements in short seeks, and increased block sizes are just some of many the complex features in modern disk drives. As the number of sophisticated disk technologies grows, the variation among disk models increases [8]. Consequently, one has a large selection of very different devices available on the market. The fact that I/O schedulers still assume a common linear disk drive



(a) 10,000 RPM 3.5" 80GB SCSI Disk

(b) 15,000 RPM 2.5" 146GB SAS Disk

Fig. 3. I/O request response time depends on the LBN distance. The empirical dependency is more complex than the one assumed by common LBN-based schedulers and is unique to specific disk drive models.

model, intuitively, should hurt LBN-based scheduling quality.

We checked how the access time depends on the LBN distance between two data blocks. Figure 3 depicts this dependency for two different devices. The regular I/O scheduler assumes that the access time increases linearly (or at least monotonically) with the LBN distance. However, from the figure we can clearly see that the dependency is not monotonous. The coefficient of linearity is 0.57 and 0.42 for the SCSI and SAS drives, respectively (1.00 corresponds to a perfectly linear dependency). Moreover, the graphs demonstrate that dependencies are different for different models.

An optimal I/O request schedule heavily depends on the specifics of the underlying hardware. Therefore, it seems reasonable to make the storage controller responsible for request scheduling. In fact, both SCSI and SATA standards support command queueing that allow the OS to submit multiple requests to the disk controller, which in turn determines the optimal request sequence [9], [10]. However, there is no way to transfer to the controller the information about desired request deadlines, which are important for real-time applications. According to our measurements, disk controllers can postpone request execution by more than 1.2 seconds if a block address is not on the optimal scheduling path. The situation is worsened by the fact that disk vendors keep their firmwares closed and the user has no control over the scheduling algorithms used within the controllers. Ironically, the only thing that the OS can do to provide more predictable service times is to disable hardware queuing entirely or flush it periodically. But in that case, disk utilization falls dramatically as the OS is unaware of drive's physical characteristics. In this work we propose to augment the OS's I/O scheduler with the knowledge of the drive's physical characteristics. This allows our OS scheduler to enforce deadlines while providing high throughput.

III. DESIGN

Section III-A explains our approach to estimate disk latencies. In Section III-B we explain how our scheduler achieves high throughput. Section III-C describes an additional algorithm that allows the scheduler to enforce deadlines of individual requests.

A. Disk Latency Estimation

A queue of N requests can be ordered in $N!$ different ways. The general task of an I/O scheduler is to pick the order that satisfies two criteria:

1) The order is the fastest when executed by a disk drive; and
2) Individual request response times are within certain limits.

The first criterion provides optimal throughput, and the second one ensures that the deadlines are met. In this paper, we argue that satisfying both criteria is hard and requires an accurate estimation of disk I/O latencies. Assume there is a function $T(o)$ that returns the execution time of some order o of N requests. One can experimentally collect the values of this function and then schedule the requests using it. Here we assume that experimentally collected values are reproducible: i.e., if the same sequence of requests is issued again, its execution time remains the same or sufficiently close. Our measurements indicate that this assumption is true for modern disk drives. For more than a 1,000,000 randomly selected orders, the deviation in execution time was within 1% for 1,000,000 iterations. However, the number of possible orders is so large that it is practically infeasible to collect latencies of all orders.

We can simplify the problem by noticing that $T(o)$ can be calculated as a sum of service times of all requests in the queue:

$$T(o) = \sum_{i=1}^{N} S_i$$

where S_i is the service time of the i-th request. S_i is a function of multiple variables, but for disk drives it depends mostly on two factors: the LBNs of the i-th and the $(i-1)$-th request. This is due to the fact that modern disk drives spend most of their time to locate the data (access time), while transfer time does not contribute much to the service time. Our experiments did not show any difference between read and write service times, but our approach tolerates potential differences [11] by using the worst service time of the two. Large I/O sizes can be addressed by logically dividing a request into smaller sizes.

So, S_i is an approximate function of two variables:

$$S_i \approx Function(LBN_i, LBN_{i-1})$$

At this point it becomes feasible to collect service times for many pairs of requests. This function can be represented as a matrix of $M \times M$ elements, where M is the number of LBN addresses that are covered by the matrix. Ideally, the matrix should cover the disk's sub-region that is accessed more frequently than the rest of the disk. Assuming that the size of on-disk "hot" dataset is 5GB, the matrix granularity is 128KB, and the size of the matrix entry is 4 bits, then the total size of the matrix is around $(5\text{G}/128\text{K})^2 \times 0.5\text{B}/2 = 400$MB. We divide by two because the matrix is symmetric, so we need to store only half of it. In our experiments, 128KB and 4 bits were enough to demonstrate significant improvements.

The matrix needs to reside in RAM or at least in a fast Flash memory; otherwise, the OS has to perform additional reads from the disk drive just to schedule an I/O request and this would make scheduling completely inefficient. 400MB is a significant amount of memory, but if one spends this 400MB on caching the data, the throughput improvement is only around 8% (400MB/5GB) for a random workload. Our scheduling can improve throughput by up to 28%, as shown in Section V. Therefore, spending expensive memory on just a cache is not justified in this case and it is wiser to use RAM for such a matrix.

A single matrix can be shared across an array of hundreds of identical disk drives (e.g., in a filer), which saves significant amount of RAM. Furthermore, some workloads prohibit caching, e.g., database writes are usually synchronous. In this case, all available RAM can be devoted to the matrix. E.g., a machine with 32GB of RAM and 100 disks can cover a dataset of 4.5TB size ($(4,500\text{G}/100/128\text{K})^2 \times 0.5\text{B}/2 = 32\text{GB}$).

However, this matrix approach scales poorly with the size of the *single* device: if a disk drive contains M blocks then the size of the matrix covering the whole disk is proportional to M^2. Another way to obtain the value of S_i is to model the drive and get S_i as the output of this model. In this case memory consumption can be significantly reduced but several other concerns emerge: how accurate is the model, how universal is it, and how high is the modeling CPU overhead. The accuracy of the model is crucial for optimal scheduling. We used DiskSim [12] to emulate several disk drives and compared the access times it predicted with the values from a pre-collected matrix. We found out that the accuracy was within 5% for 99.9% of all blocks. Our measurements also showed that CPU consumption increases by 2–3% when DiskSim is used, which we believe is a reasonable trade-off for the memory we saved.

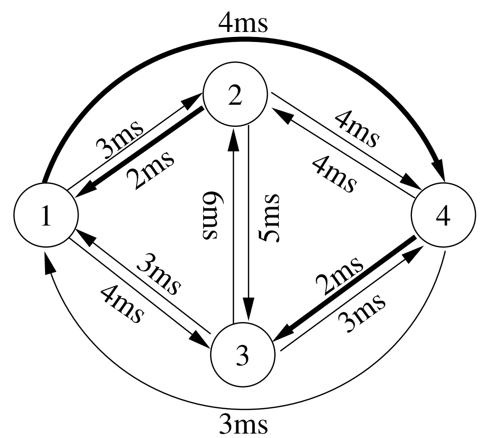


Fig. 4. Request scheduling problem as a TSP problem.

Hardware schedulers achieve high throughput using efficient reordering. Internally they use mathematical models that prove that it is possible to predict latency accurately; our work demonstrates how much benefits one can get provided that there is an accurate source of latency prediction. Our approach works for small devices, but modeling can extend it to larger ones.

B. Achieving High Throughput

When a service time for 2 consequent requests is available, the problem of scheduling to optimize disk efficiency resembles the well-known Traveling Salesman Problem (TSP). For example, assume that there are 4 requests in the queue and they are enumerated in the order they came from the applications; see Figure 4. Each request can be thought of as a vertex in a graph. The edges of the graph represent the possibility of scheduling one request after the other. As any order of requests is possible, all vertices are connected to each other (a fully connected graph). An edge's weight represents the time to service one request after the other. To find the optimal order of requests one needs to find the shortest path that covers all vertices. From Figure 4 we can see that although the requests come in the order 1-2-3-4, it is more optimal to execute them in the order 2-1-4-3, because it constitutes the shortest path.

It is well known that finding an exact solution to a TSP is exponentially hard. We initially implemented a scheduler that solves TSP exactly, but as expected it did not scale well with the queue size. There are a number of approximation algorithms that solve TSP. We picked an algorithm that is quick, easy, and provides reasonable precision, the Nearest Insertion Algorithm [13]. It works as follows:

1) Initialize the Shortest Path Edges set SPE and the Shortest Path Vertices set SPV to the empty sets.
2) For each graph vertex V_i that is *not* in the SPV:
    a) For each edge V_jV_k in the SPE set, calculate the SPE path increase $I_{V_jV_k}$ if the edge V_jV_k is

replaced by the V_jV_i and V_iV_k edges:

$$I_{V_jV_k} = W_{V_jV_i} + W_{V_iV_k} - W_{V_jV_k}$$

where W_{edge} is the weight of the $edge$.

b) For boundary vertices V_{b1} and V_{b2} in the SPV set (i.e., the vertices that have less than two adjacent edges in the SPE), calculate the path increases I_{b1} and I_{b2} if edges V_iV_{b1} and V_jV_{b2} are added to SPE, in order:

$$I_{b1} = W_{V_iV_{b1}},\ I_{b2} = W_{V_jV_{b2}}$$

Only one boundary vertex exists in the very first cycle of this loop.

c) Pick the smallest one among $I_{V_jV_k}$, I_{b1}, I_{b2} and add the corresponding edge (V_jV_i, V_iV_{b1}, or V_jV_{b2}) to the SPE set.

d) Add V_i to the SPV set.

3) When all graph vertices are in SPV, the SPE set contains an approximate solution of the TSP.

The complexity of this algorithm is $O(N^2)$, which is reasonable even for a relatively long queue. Queues that are longer than 256 requests are rare in real servers because they dramatically increase the wait time [14]. The worst case approximation ratio of the described algorithm is 2 (i.e., the resulting path might be twice longer than the optimal one). Although there are algorithms with better approximation ratios, they are much more difficult to implement, their running time is worse for small graphs, and they are often not space-efficient. The Nearest Insertion Algorithm is considered to be the most applicable one in practice [15].

In order for a TSP solution to be optimal for a real disk, an accurate service time matrix is required. Our scheduler does not require a special tool to collect this information. It collects the matrix as it runs, inferring the latency information from the requests submitted by the applications. Initially the map is empty and the scheduler orders the requests so that it can fill empty cells. For example, if there are requests $rq1$, $rq2$, and $rq3$ in the queue and the map contains information about the service time for only the $(rq1, rq2)$ and $(rq2, rq3)$ pairs, but no information about the $(rq1, rq3)$ pair, our scheduler schedules the request $rq3$ after $rq1$. As more and more matrix cells are filled with numbers, and the model becomes more precise, scheduling becomes more efficient. We demonstrate this behavior in Section V.

C. Deadlines Enforcement

More accurate estimation of disk I/O latencies allows our reordering algorithm to provide high performance without the need to keep large internal disk queues. This allows a real-time scheduler using such a reordering scheme to tightly control request service times while maintaining high efficiency. This section describes how we extended our I/O scheduler described earlier to support the notion of deadlines and guarantee request completion times as long as device's throughput allows that.

Let us formulate this problem in terms of the graph theory as we did in the previous section. Again, we have a fully connected graph with N vertices which represent requests in the queue. All edges of the graph are weighted in accordance with the service time, $W_{V_iV_j}$. In addition to that there is a deadline D_{V_i} for each vertex V_i. Deadlines are measured in the same time units as the weights of the edges. A traveling salesman should visit every vertex in the graph and it is important for him to be in certain vertices within the certain deadlines.

Assume that the salesman has picked some path through the vertices. Then it will visit each vertex V_i at specific time C_i (completion time). We call the *overtime* O_{V_i} the time by which the salesman was late at vertex V_i:

$$O_{V_i} = max(0, C_{V_i} - D_{V_i})$$

The problem of TSP with deadlines, or I/O scheduling with guarantees, is to find a sequence of vertices $V_1V_2...V_i...V_N$, such that:

$$\max_i(O_{V_i})) \rightarrow min \tag{1}$$

$$\sum_{i=1}^{i=N-1} W_{V_iV_{i+1}} \rightarrow min \tag{2}$$

Equation (1) expresses the fact that no overtime, or minimal overtime, should be found. Equation (2) states that the path should be minimal. The order of these requirements is important: we first guarantee minimal overtime; then, among the remaining solutions, we pick the one that provides the shortest path. If the system is not overloaded, overtime will be zero for all vertices, so the algorithm enforces hard deadlines. Notice, however, that this is under the assumption that estimated latency matrix is accurate enough. Our scheduler keeps updating the values in the matrix as it works. It stores only *the worst* time it has ever observed for a pair of LBN addresses. This allows to enforce deadlines with a very high probability. According to our experiments, after 100 measurements the probability to observe an even worse service time is less than $10^{-6}\%$. Storing only the worst time also addresses potential problems with the very fast accesses to the disk cache hits. Updating the values in the matrix makes our scheduler adaptive to the changes in the device characteristics, which happens, for example, when bad blocks are remapped.

This problem is proven to be NP-complete [16]. We developed an approximation algorithm to solve it. The classic method to solve deadline scheduling is the Earliest Deadline First (EDF) algorithm. It simply executes requests in the deadline order. EDF provides minimal overtime, but does not take into account service time variation among requests and consequently does not find an optimal throughput (i.e., it does not pick the shortest path in terms of the graph). Somasundara et al. solved a similar problem for mobile-element scheduling in sensor networks [16]. However, there are two important differences between mobile-element scheduling and I/O request scheduling. First, every I/O request should be eventually serviced even if it cannot meet its deadline. Second, once the request is serviced, it is removed from the original set and there is no deadline update. We merged the EDF and

Disk Model	Interf.	Cap. (GB)	RPM	Avg Seek (ms)	HBA
3.5” Maxtor 6Y200P0	PATA	200	7,200	9.3	ServerWorks CSB6
3.5” Seagate ST380013AS	SATA	80	7,200	8.5	Intel 82801FB
3.5” Seagate ST373207LW	SCSI	73	10,000	4.9	Adaptec 29320A
2.5” Seagate ST9146852SS	SAS	146	15,000	2.9	Dell PERC 6/i

TABLE I
DEVICES USED IN THE EVALUATION

the *k-lookahead* algorithm by Somasundara et al. to provide deadline enforcement in our scheduler. Our algorithm operates as follows:

1) Sort vertices by the deadline in the ascending order.
2) Pick the first k vertices from the sorted list L.
3) For all permutations $P_l = (V_1 V_2 ... V_k)$ of k vertices:
   a) Calculate the maximum overtime M_{P_l} for P_l:
   $$M_{P_l} = \max_{1..k}(O_{V_i})$$
   b) Calculate the sum of weights S_{P_l} for P_l:
   $$S_{P_l} = \sum_{i=1}^{i=k-1} W_{V_i V_{i+1}}$$
4) Among all P_l ($l = 1..k!$), pick the permutations that minimize M_{P_l}.
5) If there are multiple permutations with the same minimal M_{P_l}, then pick the case for which S_{P_l} is minimal.
6) Add the first vertex in the selected permutation P_l to the final sequence F and remove this vertex from the sorted list L.
7) Repeat Steps 2–6 if there are still vertices in L.
8) At the end, the final sequence F contains an approximate solution of the problem.

This algorithm looks through permutations of k vertices and picks the next vertex to follow among them. Because k nodes make $k!$ permutations, the overall running time of the algorithm is $O(Nk!)$. The approximation ratio in this case is $O(N/k)$. There is a trade-off in selecting the value of the constant k. If one sets k to a large value, the precision of the algorithm becomes higher (i.e., when $k = N$ the solution is absolutely optimal). However, CPU consumption grows with k. Our experiments showed that the increase of k value beyond 4 does not yield benefits, so our scheduler sets k to 4 by default, but this can be tuned.

IV. IMPLEMENTATION

We implemented our matrix-based schedulers in Linux kernel version 2.6.33. Linux has a convenient plugable interface for I/O schedulers, so our code conveniently resides in a single C file of less than 2,000 LoC. We also wrote several tools for matrix analysis which total to about 1,000 LoC. All sources can be downloaded from the following URL: https://avatar.fsl.cs.sunysb.edu/groups/mapbasedioscheduler/. Our scheduler exports a number of tunable parameters through the sysfs interface:

```
# ls /sys/block/sdb/queue/iosched/
latency_matrix
deadlines
lookahead_k
timesource
```

Servers are occasionally rebooted for maintenance and because of the power outages. We incorporated the ability to save and restore the matrix in our disk scheduler through the `latency_matrix` file. This feature also helps the user to shorten the scheduler's learning phase. If one has a matrix for some disk drive, then it makes sense to reuse it on the other machines with identical drives. If there are differences in the device characteristics, they will be automatically detected by the scheduler and the matrix will be updated accordingly.

Linux has system calls to assign I/O priorities to processes and we used those to propagate deadline information to our scheduler. The mapping between priority levels and deadlines is loaded through the `deadlines` file. The parameter k for the lookahead algorithm is set through the `lookahead_k` file. We used two time sources: a timer interrupt and the RDTSC instruction. One can set up the time source through the `timesource` file. In our experiments the RDTSC time source provided better results due to its higher precision.

We also implemented several other LBN-based scheduling algorithms to evaluate our scheduler against. We provide details in Section V-B. To experiment with hardware scheduling we modified a few HBA drivers so that one can explicitly set the size of the hardware's internal queue.

V. EVALUATION

Section V-A details the hardware we used. In Section V-B we summarize the schedulers we evaluated. In Section V-C we demonstrate that accurately estimated latency map allows to achieve better throughput than LBN-based OS I/O schedulers. Finally, in Section V-D we show that hardware scheduling does not provide adequate deadlines support, while our scheduler does.

A. Devices

We experimented with several disks to evaluate our scheduler. Table I lists the key characteristics of the drives and HBAs we used. We picked devices that differ by interface, form factor, capacity, and performance to ensure that our scheduler is universal. We present the results for the SCSI disk only, but our experiments found approximately the same degree of improvements and the behavioral trends on all of the listed devices.

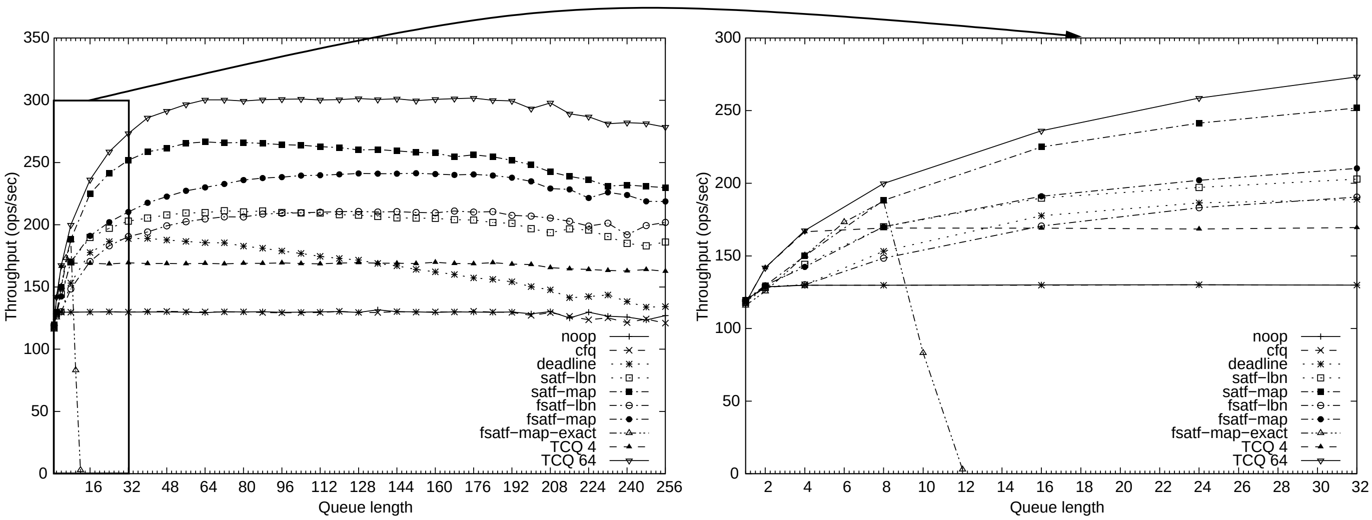


Fig. 5. Throughput depending on the queue length for different schedulers. We show the graph for 1–256 requests on the left. We zoom into the 1–32 range of requests on the right.

B. Schedulers

We picked several schedulers to compare against ours:

1) We implemented a SATF-LBN scheduler that calculates access times purely by the LBN distance and uses SATF (Shortest Access Time First) policy for scheduling. SATF is known to provide the best possible throughput among all scheduling policies [17].
2) The FSATF-LBN scheduler is identical to SATF-LBN but it freezes the queue during each dispatch round, meaning that the new incoming requests are sorted in a separate queue (FSATF). This algorithm is important because unlike SATF, it prevents postponing requests indefinitely.
3) The first variation of our scheduler, SATF-MAP, implements the algorithm described in Section III-B. It finds the shortest path using the latencies stored in the map.
4) The second variation of our scheduler, FSATF-MAP, is identical to SATF-MAP but uses a FSATF freeze policy.
5) The FSATF-MAP-EXACT scheduler solves the corresponding TSP problem exactly, without approximation.
6) NOOP is a slightly modified version of Linux's NOOP scheduler that uses pure FCFS policy (without modifications it performs sorting by LBN addresses). It serves as a baseline for the results of the other schedulers.
7) TCQ4 is hardware scheduling with a queue length of 4. We confirmed that the selection of the OS I/O scheduler does not matter in this case. In fact, the hardware completely ignores the request order enforced by the OS and applies its own ordering rules.
8) TCQ64 is the same as TCQ4, but the queue length is 64.
9) For the sake of completeness we also include the results of CFQ and DEADLINE schedulers. The Linux CFQ scheduler is the default one in most Linux distributions, so its performance results would be interesting for a lot of users. The Linux DEADLINE scheduler is often recommended for database workloads. It uses the SCAN policy but also maintains an expiration time for each request. If some request is not completed within its expiration time, the scheduler submits this request immediately, bypassing the SCAN policy.

C. Throughput

In this section, we evaluate the efficiency of our reordering algorithm. We used Filebench to emulate random-like workloads [18]. Filebench allows us to encode and reproduce a large variety of workloads using its rich model language. In this experiment, N processes shared the disk and each process submitted I/Os synchronously, sending next I/O after the previous one completed. We varied the number of processes from 1 to 256, which changed the scheduler's queue size accordingly (since each process had one outstanding request at a time). Processes performed random reads and writes covering the entire disk surface.

To speed-up the benchmarking process (in terms of matrix collection), we limited the number of I/O positions to 10,000. We picked positions randomly and changed them periodically to ensure fairness. We set the I/O size to 1–4KB, which corresponds to Filebench's OLTP workload. We set the matrix granularity to 128KB and collected performance numbers when the matrix was filled.

Figure 5 depicts how the throughput depends on the queue length for different schedulers. The left figure shows the results for the queue lengths from 1–256 and the right one zooms into the results for 1–32 queue lengths.

The NOOP scheduler's throughput does not depend on the queue length and is equal to the native throughput of the disk: slightly higher than 130 IOPS. This performance level corresponds to the situation when no scheduling is done. CFQ's throughput is identical to NOOP's, because each request is submitted synchronously by a separate process (as it is common in database environments). CFQ iterates over the list of processes in a round-robin fashion, servicing only the requests corresponding to the currently selected process. If there is only a single request from a currently selected process, CFQ switches to the next process. For synchronous processes this effectively corresponds to NOOP: dispatch requests in the order they are submitted by the applications. The DEADLINE scheduler uses the SCAN policy based on the LBN distance. Consequently, its throughput is up to 50% higher compared to NOOP and CFQ. However, when requests pass a certain expiration time (500ms by default), it starts to dispatch requests in FCFS order. This is seen in the graph: after a certain queue length, the line corresponding to DEADLINE starts to approach NOOP's line.

As expected, the SATF-LBN and FSATF-LBN schedulers exhibit better throughput compared to the previously discussed schedulers. Specifically, SATF-LBN's throughput is the best one that scheduling algorithms can achieve if they use LBN distance solely as the access-time metric. For queues shorter than 100 requests, SATF-LBN outperforms FSATF-LBN, because SATF-LBN inserts requests in the live queue, allowing more space for optimal reordering. However, with longer queues, SATF-LBN needs to perform more sorting than FSATF-LBN, and this causes SATF-LBN to perform worse.

The SATF-MAP scheduler, which implements the same SATF policy as SATF-LBN, but uses the matrix to solve the problem, performs up to 28% better. This is where the value of the latency matrix is seen: the better knowledge of the access times allows us to perform more optimal scheduling.

We implemented a FSATF-MAP-EXACT scheduler that finds the exact optimal solution of the TSP problem. As expected, its performance did not look appealing. When the queue length reaches 8 requests, its throughput drops rapidly because of the exponential complexity of the algorithm. Approximate solutions to the TSP problem performed well. The FSATF-MAP scheduler was up to 17% better than FSATF-LBN, and 13% better on average.

Finally, hardware-implemented algorithms' performance depends on the controller's internal queue length. TCQ4 provides higher throughput compared to not scheduling requests at all (NOOP), but does not outperform other OS-level I/O schedulers. TCQ64's throughput is higher than any other scheduler. Our scheduler cannot reach this level because there is an inherent overhead in submitting one request at a time compared to giving multiple requests to the scheduler at once. We believe this can be addressed by allowing the controller to accept multiple requests at a time, but forcing it to preserve the order. Although it should be possible by setting a *ordered queue tag* on every request in the SCSI queue, it did not work for our controller. Block trace analysis revealed that the controller ignores this flag. Although TCQ64 provides high throughput, it is impossible to guarantee response times. In Section V-D we discuss this problem in more details.

To demonstrate the adaptive nature of our scheduler, we collected the behavior of its throughput over time (Figure 6). It is noticeable that in the beginning of the run the scheduler's performance is relatively low because it picks the request orders that lead to completion of the matrix, not the orders that

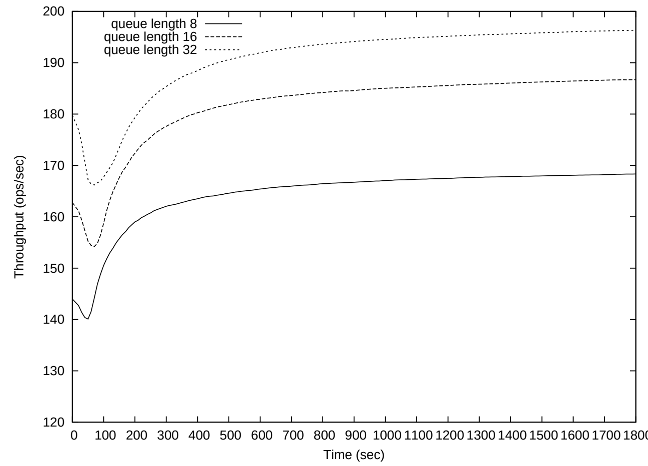


Fig. 6. Adaptive behavior of our scheduler: performance improves with time. Note: the Y axis starts at 120 ops/sec.

optimize throughput. As the matrix gets filled with the latency numbers, throughput starts to improve. Interesting is a hollow in the beginning of the graph. This happens because when the matrix is mostly empty, the scheduler reorders requests by LBN while still trying to extract the required information, which hurts performance temporarily. As time passes, the number of request sequences that have ordered LBNs (and still are not in the matrix) decreases, which leads to the drop in the throughout. After the matrix is filled with missing values, the throughput starts to grow and surpasses the original number.

Thus far we discussed random workloads, but what about workloads that are more sequential? Sequential workloads are characterized by larger request sizes or requests that arrive in serial order. When the request size is around 400KB, transfer times reach the same magnitude as access times. If two subsequent requests are adjacent to each other, access time becomes almost zero. All that make I/O reordering less effective because it only optimizes access time. This is true for all disk schedulers, including ours. However, our scheduler does not hurt performance of sequential workloads, so keeping the scheduler enabled for sequential or mixed workloads is completely valid.

D. Deadlines Enforcement

To show that our scheduler from Section III-C can efficiently enforce request deadlines, we designed the following experiment in Filebench [18]. We created 8 low-priority and 8 high-priority threads. All requests submitted by the low-priority threads were assigned 200ms deadline. We set the deadline for requests from the high-priority threads to 100ms. We collected the maximum response time and throughput for each thread at the user level. Table II shows the results for four schedulers: Earliest Deadline First (EDF), Guaranteed matrix (G-MATRIX), and hardware schedulers with queue sizes 4 and 64. We present the results of hardware schedulers to demonstrate that they cannot guarantee response times,

Sched.	Max response (ms)		Aggr. Thrpt. (ops/sec)
	100ms deadline	200 ms deadline	
EDF	99	198	130
G-MATRIX	86	192	172
TCQ4	407	419	169
TCQ64	955	1,272	236

TABLE II
RESPONSE TIMES FOR LOW (200MS) AND HIGH (100MS) PRIORITY THREADS

though show a good throughput. We do not present the results for other OS I/O schedulers because those schedulers were not designed to enforce deadlines, so their poor results are expected.

EDF enforces deadlines well: maximum response time is always lower than the deadline. This is how it should be because EDF is the most optimal algorithm if the overtime is the only optimization criterion. However, it does not optimize for throughput. Our G-MATRIX scheduler also enforces the deadlines, but its throughput is 32% higher. The hardware schedulers ignore the order in which the OS submits requests and violates the deadlines. As you can see from the table, the maximum response time for TCQ4 is twice over the deadline and 6–9 times worse for TCQ64. By breaking the deadlines, TCQ64 significantly improves throughput (by 37% compared to G-MATRIX). We explained this phenomenon earlier in Section V-C.

VI. Related Work

Improving disk efficiency was a main focus of early disk scheduling algorithms, that observed that ordering of I/O requests can significantly improve disk performance. Most of the algorithms were designed to minimize the disk head movement to reduce seek times, such as SCAN [19] and Shortest Seek Time First (SSTF), while other algorithms tried to minimize total positioning delays by minimizing rotational latencies as well as seek times [14], [20]. Coffman et al. analyzed the Freezing SCAN (FSCAN) policy compared to FCFS, SSTF, and SCAN [21]. The FSCAN scheduler freezes the request queue before the start of each arm sweep. This improves response time and fairness compared to SCAN. Hefri performed extensive theoretical and simulation-based analysis of FCFS and SSTF, showing that SSTF exhibits the best throughput under almost all workloads but its response time variance can be large, delaying some requests by a substantial amount of time [17]. Geist et al. presented a parameterized algorithm that represents a continuum of scheduling algorithms between SSTF and SCAN [22].

All the studies mentioned above assumed that a scheduling algorithm has access to the detailed physical characteristics and current state of the drive. Since modern hard drives hide their internal details and expose only a limited Logical Block Number (LBN) interface, these algorithms had to be implemented in the disk controller's firmware, which is only possible by drive vendors. Our scheduling approach brings detailed device characteristics to the upper layer so that better scheduling can be performed by the OS. Due to the closed-source nature of disk drives' firmware, researchers mostly used simulators (such as DiskSim [12] or specially written ones) or theoretical calculations to demonstrate the performance of their algorithms. All the experiments in this paper, however, were conducted on real hardware without emulation.

Other disk schedulers optimized disk performance by using LBN-based approximation of seek-reducing algorithms [23]. Linux is the richest OS in terms of I/O schedulers, and it includes noop, anticipatory, deadline, and CFQ schedulers. All of these schedulers rely on the regular LBN interface [24]. Our reordering scheme is based on more accurate information about disk I/O latencies and it is more efficient than LBN-based approaches.

Many real-time schedulers aim to optimize performance while meeting real-time guarantees by combining a reordering algorithm to optimize disk efficiency, such as SCAN, with Earliest Deadline First (EDF) real-time scheduling. Some of them use LBN-based reordering [6], [5] and others rely on the detailed knowledge of the disk [25]. Our approach for accurate I/O latency estimation and our reordering scheme is complementary to many of these scheduling algorithms, and can be used to improve overall disk efficiency in these schedulers.

Reuther et al. proposed to take rotational latency into account for improving performance of their real-time scheduler [4]. The authors used a simplified disk model and as a result they were only able to calculate *maximum* response times. The implementation Reuther et al. had was for the Dresden Real-Time Operating System [26]. Our implementation is for much more common Linux kernel and consequently we were able to compare our scheduler to other schedulers available in Linux. Michiels et al. used the information about disk zones to provide guaranteed throughput for applications [27]. However, they were mainly concerned about throughput fairness among applications, not response time guarantees. Lamb et al. proposed to utilize the disk's rotational latency to serve I/O requests from background processes [28]. This is another way to increase disk utilization, providing high throughput while enforcing response time deadlines [29].

Yu et al. conducted the study similar to ours [30]. They examined the behavior of several Linux I/O schedulers running on top of a command-queuing capable device. Their analysis provided the evidence of possible redundant scheduling, I/O starvation, and difficulties with prioritizing I/O requests when command queuing is enabled. The authors also proposed a mechanism for overcoming these problems. Their idea is to switch command queuing on and off depending on the value of a specially developed metric. The disadvantage of their approach is that the metric has a number of tunable parameters which are difficult to set appropriately. Moreover, the appropriate values depend on the hardware specifics. Conversely, our approach is simpler and more general: it moves all scheduling decisions to the OS level, works for virtually any device and performs all tuning automatically. The CPU usage of our solution is negligible because we use computationally lightweight approximation algorithms, and our memory usage can be limited by the administrator.

VII. Conclusions

Hardware schedulers are capable of providing excellent throughput but a user cannot control response times of individual requests. OS schedulers can strictly enforce response time deadlines but their throughput is significantly lower than what a disk can provide. The ability to estimate disk latencies at the OS level allows to achieve higher throughput while enforcing the deadlines. We designed and implemented an I/O scheduler that collects a matrix of service times for and underlying disk drive. It then performs request scheduling by finding an approximate solution of a corresponding TSP problem. The design of our scheduler incorporates a number of trade-offs between CPU usage, memory usage, universality, and simplicity. The scheduler does not require a distinct learning phase: it collects hardware information on the fly and performs better as more information becomes available. We successfully tested our scheduler on a variety physical disks and showed it to be up to 28% more efficient than other schedulers. Compared to hardware level scheduling solutions, our scheduler enforces deadlines as requested by the processes.

VIII. Future Work

We plan to work towards memory footprint reduction in our scheduler. We believe that pattern recognition techniques are especially promising in this respect because latency matrices contain a lot of fuzzy patterns which regular compression algorithms cannot detect. We plan to work on extending our scheduler to a wider set of devices, specifically solid-state drives and hybrid devices. We expect that matrix design will require modifications to reflect storage devices with significantly different hardware characteristics. Virtualization is another direction for future work, because virtualization layers tend to further perturb, or randomize, I/O access patterns. We successfully tested a prototype of our scheduler inside a virtual machine and the scheduler was capable of detecting significant latency differences in the underlying storage.

References

[1] A. Gulati, C. Kumar, and I. Ahmad, "Storage workload characterization and consolidation in virtualized environments," in *Proceedings of 2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[2] D. Sears, "IT Is Heavily Invested in ERP, Application Consolidation Rising," 2010, www.eweek.com/c/a/IT-Management/IT-Is-Heavily-Invested-in-ERP-Application-Consolidation-Rising-244711/.

[3] C. Li, G. Peng, K. Gopalan, and T. cker Chiueh, "Performance guarantee for cluster-based internet services," in *in The 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2003.

[4] L. Reuther and M. Pohlack, "Rotational-position-aware real-time disk scheduling using a dynamic active subset (das)," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium*, 2003.

[5] A. Povzner, T. Kaldewey, S. Brandt, R. Golding, T. M. Wong, and C. Maltzahn, "Efficient guaranteed disk request scheduling with fahrrad," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, ser. Eurosys '08. New York, NY, USA: ACM, 2008, pp. 13–25. [Online]. Available: http://doi.acm.org/10.1145/1352592.1352595

[6] A. L. N. Reddy and J. Wyllie, "Disk scheduling in a multimedia i/o system," in *Proceedings of the first ACM international conference on Multimedia*, ser. MULTIMEDIA '93. New York, NY, USA: ACM, 1993, pp. 225–233. [Online]. Available: http://doi.acm.org/10.1145/166266.166292

[7] M. J. Stanovich, T. P. Baker, and A.-I. A. Wang, "Throttling on-disk schedulers to meet soft-real-time requirements," in *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*, ser. RTAS '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 331–341. [Online]. Available: http://dx.doi.org/10.1109/RTAS.2008.30

[8] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer, "Benchmarking file system benchmarking: It *is* rocket science," in *Proceedings of HotOS XIII:The 13th USENIX Workshop on Hot Topics in Operating Systems*, Napa, CA, May 2011.

[9] B. Dees, "Native Command Queuing - Advanced Performance in Desktop Storage," in *Potentials*. IEEE, 2005, pp. 4–7.

[10] "Tagged Command Queuing," 2010, http://en.wikipedia.org/wiki/Tagged_Command_Queuing.

[11] C. Ruemmler and J. Wilkes, "An introduction to disk drive modeling," *IEEE Computer*, vol. 27, pp. 17–28, 1994.

[12] J. S. Bucy and G. Ganger, *The DiskSim Simulation Environment Version 3.0 Reference Manual*, 3rd ed., January 2003, www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-CS-03-102.pdf.

[13] D. Rosenkrantz, R. Stearns, and P. Lewis, "Approximate Algorithms for the Traveling Salesperson Problem," in *15th Annual Symposium on Switching and Automata Theory (SWAT 1974)*. IEEE, 1974, pp. 33–42.

[14] M. Seltzer, P. Chen, and J. Ousterhout, "Disk Scheduling Revisited," in *Proceedings of the Winter Usenix*, 1990.

[15] A. Frieze, G. Galbiati, and F. Maffioli, "On the Worst-case Performance of Some Algorithms for the Asymmetric Traveling Salesman Problem," *Networks*, 1982.

[16] A. Somasundara, A. Ramamoorthy, and M. Srivastava, "Mobile Element Scheduling for Efficient Data Collection in Wireless Sensor Networks with Dynamic Deadlines," in *Proceedings of the 25th IEEE Real-Time Systems Symposium*. IEEE, 2004, pp. 296–305.

[17] M. Hofri, "Disk Scheduling: FCFS vs. SSTF revisited," *Communication of the ACM*, vol. 23, no. 11, November 1980.

[18] "Filebench," http://filebench.sourceforge.net.

[19] P. Denning, "Effects of Scheduling on File Memory Operations," in *Proceedings of the Spring Joint Computer Conference*, 1967.

[20] D. Jacobson and J. Wilkes, "Disk Scheduling Algorithms based on Rotational Position," Concurrent Systems Project, HP Laboratories, Tech. Rep. HPLCSP917rev1, 1991.

[21] E. Coffman, L. Klimko, and B. Ryan, "Analysis of Scanning Policies for Reducing Disk Seek Times," *SIAM Journal on Computing*, vol. 1, no. 3, September 1972.

[22] R. Geist and S. Daniel, "A Continuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems (TOCS)*, vol. 5, no. 1, February 1987.

[23] B. Worthington, G. Ganger, and Y. Patt, "Scheduling Algorithms for Modern Disk Drives," in *Proceedings of the ACM Sigmetrics*, 1994.

[24] J. Axboe, "Linux Block I/O — Present and Future," in *Proceedings of the Ottawa Linux Symposium*, 2004.

[25] H.-P. Chang, R.-I. Chang, W.-K. Shih, and R.-C. Chang, "Gsr: A global seek-optimizing real-time disk-scheduling algorithm," *J. Syst. Softw.*, vol. 80, no. 2, pp. 198–215, February 2007. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2006.03.045

[26] H. Hrtig, R. Baumgartl, M. Borriss, C. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schnberg, and J. Wolter, "Drops: Os support for distributed multimedia applications," in *Eighth ACM SIGOPS European Workshop*, 2003.

[27] W. Michiels, J. Korst, and J. Aerts, "On the guaranteed throughput of multi-zone disks," *IEEE Transactions on Computers*, vol. 52, no. 11, November 2003.

[28] C. R. Lumb, J. Schindler, and G. R. Ganger, "Freeblock scheduling outside of disk firmware," in *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*. Monterey, CA: USENIX Association, January 2002, pp. 275–288.

[29] Y. Zhu, "Evaluation of scheduling algorithms for real-time disk i/o," 2007.

[30] Y. Yu, D. Shin, H. Eom, and H. Yeom, "NCQ vs I/O Scheduler: Preventing Unexpected Misbehaviors," *ACM Transaction on Storage*, vol. 6, no. 1, March 2010.