Blurring the Line Between OSes and Storage Devices

Gregory R. Ganger

December 2001
CMU-CS-01-166

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Contact: Greg Ganger (ganger@ece.cmu.edu)

Abstract

This report makes a case for more expressive interfaces between operating systems (OSes) and storage devices. In today's systems, the storage interface consists mainly of simple `read` *and* `write` *commands; as a result, OSes operate with little understanding of device-specific characteristics and devices operate with little understanding of system priorities. More expressive interfaces, together with extended versions of today's OS and firmware specializations, would allow the two to cooperate to achieve performance and functionality that neither can achieve alone.*

This report consists of the technical content of an NSF proposal submitted in January 2001 and funded in June 2001 under the Information Technology Research (ITR) program. The only divergence from the original proposal is the removal of non-technical content (e.g., budgets, biographies, and results from prior NSF support).

These ideas have been developed in collaboration with the other faculty, students and staff of CMU's Parallel Data Lab (PDL). We thank the members and companies of the Parallel Data Consortium (including EMC, Hewlett-Packard, Hitachi, IBM, Intel, LSI Logic, Lucent, Network Appliances, Panasas, Platys, Seagate, Snap, Sun, and Veritas) for their interest, insights, feedback, and support.

Keywords: disk drives, storage devices, operating systems, disk scheduling, file systems, storage management.

Extended Abstract

Carnegie Mellon University proposes to explore more expressive interfaces between operating systems (OSes) and storage devices. Together with extended versions of today's aggressive OS and disk firmware specializations, more expressive interfaces will allow the two to cooperate to achieve performance and functionality that neither can achieve alone. Specifically, greater cooperation will enable:

- Device-side specializations that have access to system-provided information about priority levels and forthcoming demands. For example, we will experiment with *freeblock scheduling*, a fine-grained disk scheduling mechanism that would allow lower priority requests to make forward progress with (almost) no impact on foreground access times, even when the foreground requests always keep the disk busy.
- OS-side specializations that have access to device-provided details about physical data layout and firmware cache algorithms. For example, we will experiment with *track-aligned extents*, a file system (FS) allocation scheme that would exploit detailed disk knowledge to achieve higher bandwidth and near-zero access time variance for large requests.

For over 15 years, the OS-storage interface (e.g., SCSI or IDE) has remained static and non-expressive, consisting mainly of `read` and `write` commands. This same time period has seen much independent advancement on both sides of this interface: for example, file systems and databases carefully co-locate structures and cache disk blocks, and disk firmware aggressively schedules media accesses and prefetches data. Given the ever-growing gap between disk access times and most other computer system capabilities, storage performance continues to be critical for many computing environments. Thus, efforts continue on both sides of the interface. However, because they evolve independently, each operates with an increasingly poor understanding of the features and limitations of the other, missing significant opportunities for enhancement and even conflicting in some instances.

With more expressive interfaces, OSes and storage devices can exchange information and make more informed choices on both sides of the storage interface. For example, only the OS side has detailed information about the application sources and priorities of different disk requests. On the other hand, only the disk firmware has exact information about its internal algorithms and current state. Combining this information appropriately will allow the device to focus on what is most important to the system and will allow the OS to tune its activity to the disk firmware's strengths and avoid its weaknesses.

Achieving OS-device cooperation will require understanding of trade-offs in several areas, including: (1) ideal overall storage management algorithms given all information from both sides, (2) appropriate partitioning of policies/mechanisms across the two sides, (3) interfaces that expose the relevant information without departing far from the simplicity that has enabled the robustness of modern storage systems, and (4) specific uses of new information on each side of the interface and their associated benefits.

Getting Disks and OSes to Cooperate

1. Objectives and Significance

Storage systems have long been a source of performance and management problems, and they are growing in importance as the years pass. Of particular concern are the slow-to-improve mechanical positioning delays. To address storage's problems, storage devices and host systems have lots of resources and knowledge. Independently, engineers of both device firmware and OS software aggressively utilize their knowledge and resources to mitigate disk performance problems. The overall goal of the proposed research is to increase the cooperation between these two sets of engineers, which will significantly increase the end-to-end performance and robustness of the system. Achieving this goal will require more expressive storage interfaces to enable increased communication and new cooperative algorithms for realizing its benefits.

A fundamental problem with current storage systems is that the storage interface hides details from both sides and prevents communication. For 15 years, the same simple READ/WRITE storage interface has allowed much advancement on both sides. The result of this independent advancement is that the two sides have lost touch with each other's needs and capabilities, which in turn has led to conservatism and lost opportunities for improvement. For example, storage devices can schedule requests to maximize efficiency, but host software tends not to expose much request concurrency because the device firmware does not know about host priorities and considers only efficiency. Likewise, host software can place data and thus affect request locality in a variety of ways, but currently does so with only a crude understanding of device strengths and weaknesses, because detailed device knowledge is not available. As a third example, firmware prefetching and caching algorithms struggle to figure out access patterns and maximize the value of its on-board memory, but generally lose most benefit to overlap with data kept and prefetched by host algorithms.

All of these difficulties could be avoided by allowing the host software and device firmware to exchange information. The host software knows the relative importance of requests and also has some ability to manipulate the locations that are accessed. The device firmware knows what the device hardware is capable of in general and what would be most efficient at any given point. Thus, the host software knows what is important and the device firmware knows what is fast. This research will explore new storage interfaces and algorithms for exchanging and exploiting the collection of knowledge. Explicit cooperation will match device strengths to application needs and eliminate redundant, guess-based optimization. The result will be storage systems that are simpler, faster, and more manageable.

The remainder of this proposal motivates end-to-end storage cooperation in more detail, identifies general and specific research questions, and discusses the impact expected. Section 2 provides background on storage interfaces, advances on both sides of these interfaces, and related research and educational activities. Section 3 describes some major research issues to be addressed in realizing this cooperative vision of storage systems. Section 4 details one specific research effort, in which the data placement algorithms of file systems will be modified to better match disk characteristics. Section 5 details a second specific research effort, in which precise in-firmware disk scheduling will be extended with knowledge of host OS priorities. Section 6 discusses the impact of this work in the field and in the classroom.

2. Background and Prior Work

For the past 15 or so years, the most common storage interfaces (SCSI and IDE) have consisted mainly of the same simple read and write commands. This consistent high-level interface has enabled great portability, interoperability, and flexibility for storage devices and their vendors.

In particular, the resulting flexibility has allowed very different devices such as disk drives, solid-state stores, and caching RAID systems to all appear the same to host operating systems. In the continuing struggle to keep storage devices from becoming a bottleneck to system performance and thus functionality, system designers have developed many mechanisms for both storage device firmware and host OSes. These mechanisms have almost exclusively been restricted to only one side of the storage interface or the other. In fact, evolution on the two sides of the storage interface has reached the point where each has little idea of or input on the detailed operation of the other. We

believe that this separation, which once enabled great advances, is now hindering the development of more cooperative mechanisms that consist of functionality on both sides of the interface.

The remainder of this section discusses recent technology and interface trends, representative advances on each side of the interface, previous cooperative mechanisms, and current practice in storage system education.

2.1 The Storage Interface

A storage interface consists of both the command set (the set of command and response messages) and a physical transport (wires and protocols). The latter has evolved steadily over time, as physical technology trends enable faster and more switch-based communication – for example, SCSI transports have progressed from shared SCSI buses to FibreChannel SANs (Storage Area Networks) to current SCSI-over-IP (over Gigabit Ethernet) standardization efforts. Storage interface command sets, on the other hand, have changed very little – most systems continue to use the same handful of SCSI commands today as they did 10 years ago.

The SCSI command set includes about 30 different commands, many of which are optional or have optional variants. Most operating systems utilize only a very small subset of these commands. The most commonly used SCSI commands are READ and WRITE, which request a transfer of some number of "logical blocks." The SCSI interface exposes storage capacity as a linear sequence of such constant-size logical blocks. The relevant blocks for any command are specified by providing a start LBN (logical block number) and a count. Internally, of course, the device firmware translates LBNs to physical media locations; see Figure l. At system initialization, informational SCSI commands (such as TEST_UNIT_READY and READ_CAPACITY) are used to configure internal tables. Also, when problems arise, the REQUEST_SENSE error information fetch command is often used for diagnosis. Beyond these, most systems do not make use of the various other commands, many of which are helpful mainly to those debugging the devices or in very specialized circumstances. The IDE command set and its use in systems is quite similar to that described above for SCSI.

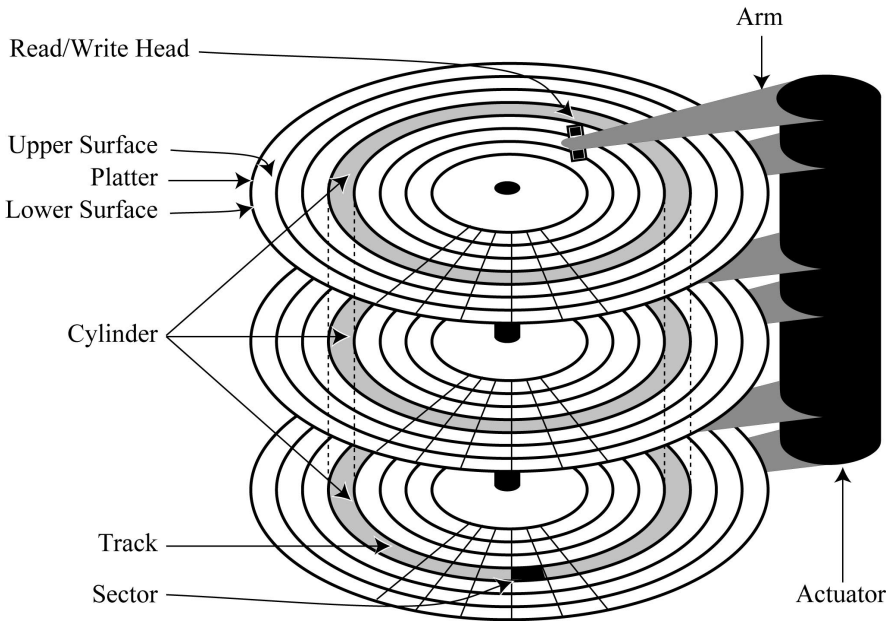

Figure 1: Disk drive internals.

As mentioned previously, this unchanging storage interface has been a mixed blessing for computer systems. On the positive side, consistency and simplicity generally yield robustness and interoperability, and this has largely been true for storage systems. Also, the well-defined interface provided freedom to implementers on both sides of the interface, enabling many changes and advances to be cleanly integrated into unaware systems. For example, ten

years ago, RAID arrays were easily added to systems as SCSI storage devices, without having to make those systems explicitly aware of the change. On the negative side, this same freedom to change has caused even the common components on each side (disk drives and host software) to lose any detailed understanding of the strengths, weaknesses, and priorities of the other. Since the interface does not provide for such communication or for detailed cooperation, it now hinders forward progress at a time when storage represents a growing performance and functionality concern.

2.2 Device-side Advances

Behind the storage interface, disk drive firmware has advanced dramatically over the past 15 years. Concrete advances can be seen in a number of areas, including LBN-to-physical mappings, caches and cache management, and detailed internal scheduling and parallelism. This section briefly describes some of these advances and their relationship to this proposal.

One place where device-side advances have departed significantly from the understanding of host software is in the mapping of LBNs to physical locations (specifically, <*cylinder,surface,sector*> triples). Once straightforward, this mapping function varies from disk to disk, involving variable numbers of physical sectors per track and unused sectors withheld for media defect management purposes. For example, most modern disks employ zoned recording (see Figure 2), in which outer cylinders store more sectors than inner cylinders [33, 45, 37]. Also, most modern disks expose fewer LBNs than could maximally be stored, using the extras to internally remap unusable sectors (i.e., those in defective media regions) to usable spares. The result of these complex mappings is that any nominal mapping assumed by host software is unlikely to accurately represent the actual physical layout. Thus, modern host software is unable to make detailed placement choices based on physical cylinder and track boundaries. As another example, BSD FFS's so-called "cylinder groups" [31] no longer bear any relationship to disks' actual cylinder boundaries.

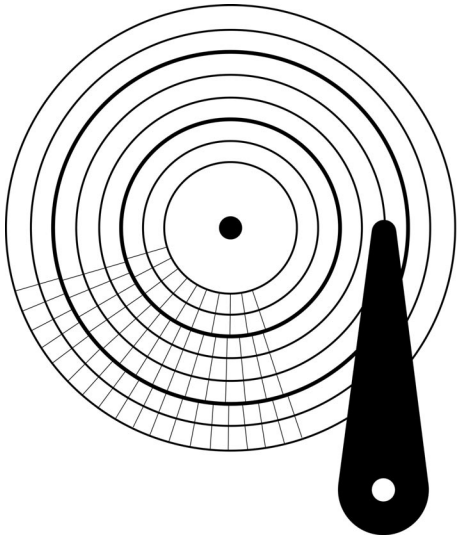

Figure 2: Top view of a disk surface with 3 zones. The innermost and outermost zones each contain 2 cylinders. The middle zone contains three.

Another advance is in the on-board memory management of disk drives [45, 37]. As memory densities have grown, multi-megabyte caches have replaced the small buffers of old. In recent years, the effectiveness of this cache memory has grown as the management algorithms have become more sophisticated. For example, physically sequential sectors are usually prefetched into this memory at near-zero overhead. Many drives also move data to/from the media in the order it passes under the read/write head rather than in strict ascending LBN order. Also, the cache space is now managed in units that reflect the expected access pattern, and many drives allow limited write-back caching. What is missing is coordination with host memory caches. The drive firmware has no

information regarding host caching policies and thus attempts to provide benefit with very little information. Since these host policies vary from system to system, drive-level caching often provides only the benefits of sequential prefetching in practice.

A third area of progress is in request scheduling [38, 46], where the drive has the most detailed information about access efficiency. As cost-effective computation power has been incorporated into disk drives, they have been able to take advantage of the computation to make precise decisions about which requests to service next. Using algorithms that minimize overall positioning time (*seek+rotation*) [48, 24], modern disks can significantly reduce average response times. The drive firmware also aggressively overlaps parts of distinct requests (e.g., bus transfer for one and media transfer for another) in order to avoid unused resources. Unfortunately, the disk firmware has no way of understanding the relative importance of different requests. As a result, many OSes and databases make their own request scheduling decisions and do not give disk firmware an opportunity to make use of its detailed knowledge. This fact is a constant source of discontent for drive firmware engineers.

2.3 OS-side Advances

Operating system designers have long struggled with disk performance, developing many approaches to reduce mechanical positioning overheads and to amortize these overheads over large media transfers. At one time, many of these techniques incorporated details of storage device operation, but they have long since lost access to these details. This section briefly describes some of the changes and advances relevant to this proposal.

Although it now also occurs within disk firmware, request scheduling has traditionally been in the purview of Oses [47, 13]. Although many OSes continue to claim ownership over request scheduling, their ability to do so effectively has been hampered by lack of information. For example, algorithms such as Shortest-Seek-Time-First are actually implemented with no direct knowledge of seek times or distances — instead, modern OSes approximate this algorithm by picking the requests that are closest in LBN space (i.e., smallest difference in LBNs). This works fairly well in practice [55], but does not allow for the more effective Shortest-Positioning-Time-First algorithms used in modern disks.

Most data placement efforts in OSes have focused on increasing the size of disk requests, amortizing positioning overheads over longer data transfers. This generally involves grouping sets of data and metadata objects into large contiguous regions, either by time (as in log-structured file systems [42]) or by name (as in co-locating file systems [15, 39]). No effort is made to match data placements to the physical boundaries of the device, since this information is not available. Previous algorithms for rotationally placing objects and considering cylinder boundaries (as in FFS's cylinder groups) are no longer operational.

Data caching efforts have focused on aggressive write-back and prefetching to hide lengthy disk access times. In particular, many schemes have been proposed for identifying access patterns [28], allowing application and compiler hinting [6, 36, 5] and even speculatively executing programs to get hints [10]. There are also application interfaces, such as dynamic sets [51] and disk-directed I/O [27] that allow groups of requests to be specified collectively. Similarly, schemes like soft updates [17] and RIO [11] allow aggressive write-back caching of data. However, in deciding what background accesses to do when, no information about what the disk could do efficiently is considered.

2.4 Previous Cross-boundary Enhancements

There do exist previous examples of cooperative enhancements of the form promoted in this proposal. Given the non-expressive standard interfaces, those that make it into production environments are usually driven by reaction to major changes in usage. For example, the rise of RAID arrays brought with it brief popularity of disks that would perform a parity update operation (read parity data, XOR with provided block, and write back parity block [7]; this mechanism reduces bus overheads and distributes parity computations, but eventually was identified as a non-critical feature. On the other hand, optionally external control over power management and recalibration remain popular features for mobile and video environments, respectively. In addition, one of the dirty little secrets of the storage industry is that ad hoc extensions are sometimes added to disk firmware for particularly large customers.

In addition to the handful of real examples, there have been a number of research efforts that propose or would benefit from more expressive storage interfaces. For example, some have proposed to dynamically place data near the disk head [12, 54] or to piggyback write-backs on rotational delays [4]. Perhaps the clearest examples are recent proposals for object-based storage [19, 18, 35] and those for Active disks [1, 41, 25]. Object-based storage defines a new storage interface that is much like a file system with a flat namespace; this does make the interface somewhat more expressive, but not nearly enough in our view. Active disks would allow application code to be downloaded and executed inside disk firmware; this represents complete expressiveness of sorts, but raises robustness and complexity issues that have kept disk vendors highly skeptical.

The research focus of this proposal is to identify more cooperative enhancements to both sides of the storage interface and corresponding extensions to it. Sections 3, 4 and 5 describe concrete examples of combining host knowledge (e.g., priorities) with disk knowledge (e.g., physical layouts) to realize improved performance, more predictable latencies, and non-interfering forward progress for background activity.

2.5 Education in Storage Systems

Interestingly, although perhaps not surprisingly, current education practices regarding storage systems mimic the independent development described above. Specifically, students are usually exposed to storage systems either via a file system segment of an operating systems class or via an I/O lecture or two in a computer architecture class. Some schools, such as Carnegie Mellon, also have storage device classes that cover magnetics, materials, and mechanics of disk drives. Each of these approaches gives students a narrow, one-sided view of storage systems. By doing so, they fail to provide students with an end-to-end appreciation of storage problems and solutions. This is a continual problem for the storage industry (one of the fastest growing computer industries; witness EMC, Network Appliance, and Veritas, among others), and for the graduating students who take jobs in the storage industry and find themselves unprepared for their work.

What is needed is a class that treats storage systems as a first-class topic. Such a class would cover the problem from one end (driving applications) to the other (device technologies), allowing students to see how all of the pieces fit together to solve real storage problems. To address the need, we are developing such a class. This class will cover the design, implementation, and use of storage systems, from the characteristics and operation of individual storage devices to the OS, database, and networking approaches involved in tying them together and making them useful. Along the way, we will examine several case studies of real systems, demands placed on storage systems by important applications, and impacts of trends and emerging technologies on future storage systems.

3. Research: Understanding When and How Cooperation Helps

The goal of cooperation between host software and device firmware raises a number of questions in need of research. At the highest level, there are two main questions: (1) what could one do if the two separate components (host and device) were one, with access to all relevant information? And (2) given that pragmatics do dictate that functionality be partitioned amongst the two components, what should be done on each and what should the interface between them look like?

Although we will always keep in mind these very general questions, our research efforts will work towards them with five more specific questions: (1) what should change in the host to better match device characteristics? (2) what should change in device firmware to better match what the host views as important? (3) how should the storage interface be extended to make these changes possible and effective? (4) how much device-specific information can be obtained and used from outside the device? (5) how much complexity and overhead is involved with extending disk firmware functionality?

3.1 Adapting Host Software to Device Characteristics

Over the years, the host-level software that manages storage devices has lost touch with the details of device mechanics and firmware algorithms. Unlike with many other components, however, these details can have dramatic, order-of-magnitude effects on system performance. For example, Ganger and Kaashoek [15] showed order-of-magnitude improvements for small file workloads from exploiting disk bandwidths to proactively avoid positioning

overheads. Looking further back, FFS's allocation algorithms originally tried to carefully avoid rotational latency delays by skip-sector placement of data blocks — this prevented the turnaround latency of two successive reads or writes from causing nearly-full rotations of delay.

Identifying specific examples where the host-level software can change in relatively small ways to better match device characteristics will represent one important step towards actually realizing greater cooperation. Section 4 describes one specific example that we will explore: track-aligned extents. Several disk drive advances in recent years conspire to make the track a sweet spot in terms of access unit, but it only works when accesses are aligned on and sized to track boundaries. Accomplishing track-aligned extents in file systems will require several changes, including techniques for identifying the boundaries and file system support for variable sized allocation units. If this is done, our preliminary examination suggests that track-aligned extents can provide significant performance and predictability benefits for large file and video applications.

3.2 Adapting Device Firmware to Host Priorities

Disk firmware includes aggressive cache management and request scheduling facilities that go largely unused in many environments. This unfortunate circumstance arises from the fact that the device views all requests as equally important, whereas this is simply not true in the overall system [16]. For example, some requests block important processes and others are strictly in the background. If the firmware understood this, it could be trusted to use its detailed local knowledge appropriately; in today's systems, however, the firmware would put efficiency ahead of system priorities because it does not know about the latter.

With minor extensions to the current storage interface, it should be possible to convey simple priority information to the device firmware. The question then becomes how to utilize this information in firmware algorithms. Section 5 describes one specific mechanism that we will explore for using this information: freeblock scheduling. By accurately predicting the rotational latencies of high-priority requests, it becomes possible to make progress on background activity with little or no impact on foreground request access times. Our preliminary examination indicates that this can increase media bandwidth utilization by an order of magnitude and provide significant end-to-end performance improvements.

3.3 Evolving the Storage Interface to Support Cooperation

Realizing the cooperative vision of this proposal will require more expressive interfaces for storage devices. Even the one-sided changes indicated above require more information to flow in one direction or another. We further plan to explore interfaces and algorithms that allow even more cooperation between host software and device firmware. For example, we envision an interface that would allow the host system to direct the device to write a block to any of several locations (whichever is most efficient); the device would then return the resulting location, which would be recorded in the host's metadata structures. Such an interface would allow the host (e.g., database or file system) and the device to collaborate when making allocation decisions, resulting in greater efficiency but no loss of host control.

To experiment with new interfaces, we will build a storage device emulator atop emerging SCSI-over-IP subsystems. Similar to how standardized network file system interfaces allowed for appliance-like servers, SCSI-over-IP will allow us to emulate a storage device with another computer system. We will integrate an accurate disk simulator and any experimental algorithms into this emulator. Together with source code to the original host OS, this setup will give us as much reality as possible without being inside real disk firmware. We also plan to work with disk manufacturers to experiment with promising approaches in real disk firmware.

3.4 Using Device-specific Knowledge from Outside the Device

An important practical research question that must be answered in pursuing this vision is how much device-specific knowledge and low-level control is available from outside the device firmware. In particular, if complete knowledge and control is available externally, then it may be unnecessary for host software to cooperate with device — instead, the host software can directly do exactly what it needs done, bypassing the firmware engineers entirely. Although we do not believe complete control to be possible, it is important to understand how close one can get. This understanding does not yet exist.

There has been some progress towards this understanding. For example, algorithms have been designed for extracting device characteristics from host software [56] and even for automatically identifying various firmware algorithms [44]. Two recent attempts to accurately predict overall access times and thus schedule requests have given partial answers. Specifically, Yu et al. [57] were able to very accurately predict rotational position, but were much less able to predict overall access times when seeks were involved. Huang and Chiueh [22] were able to construct a similar scheduler with similar results. Both studies focused on just a single disk with the firmware's caching and scheduling algorithms disabled. What continues to elude us is a comprehensive understanding of what can be done from outside the device.

3.5 Understanding Issues Involved with Extending Disk Firmware Functionality

An equally important question relates to the practical limitations involved with working within disk firmware, for example those related to ASIC interactions, timeliness requirements, and limited runtime support. In some sense, this is not deep research, since disk manufacturers have been extending firmware for years. However, it is a critical practical consideration for any work that proposes extensions; unfortunately, this understanding has been largely absent outside of these organizations. In addition to refining some of our research, this understanding will transfer directly into the classroom.

We plan to gain this understanding by working with disk manufacturers to develop in-drive prototypes of promising new interfaces and algorithms. Building such prototypes will enable both the understanding and the direct experimentation with new storage interfaces and firmware algorithms that are needed to identify their real value. Historically, disk manufacturers have not been comfortable sharing their firmware source code (and still are not), but both Seagate and Quantum (now Maxtor) have invited our students to intern with them and pursue this avenue.

4. Example#1: Track-aligned Extents

Modern host software (e.g., file systems or databases) performs aggressive on-disk placement and request coalescing, but generally does so with only vague view of device characteristics — generally, the focus is on the notion that "bigger is better, because it amortizes positioning delays over more data transfer." However, there do exist circumstances where considering specific boundaries can make a big difference. Track-aligned extents is a new approach to matching system workloads to device characteristics and firmware enhancements. By placing and accessing largish data objects on track boundaries, one can avoid most rotational latency and track crossing overheads. This section briefly describes track-aligned extents, the changes required to utilize them, the resulting benefits, and preliminary results indicating great promise.

4.1 Why Track-aligned Extents

Recent disk technology trends conspire to make the track an ideal unit of allocation and access. Specifically, accessing a full track's data has two significant benefits:

Avoiding track switches: A single track is the maximum amount of data that can be accessed without a track switch after the initial positioning delay. This is important because a track switch requires almost as much time as any other short seek. Further, this is expected to continue to be true in future years, because track density increases continue to cause settling times to remain relatively constant over the years. As a result, track switches become increasingly expensive relative to other parts of media accesses. For example, the recent Seagate 15K RPM disk [46] has a track switch time of 0.8ms, which represents about 1/4 of a rotation.

Eliminating rotational latency: When accessing a full track, rotational latency can be eliminated by modern disk firmware. Specifically, the so-called "immediate access" (a.k.a. "zero-latency access") feature of modern disks accesses sectors on the media in the order that they pass under the read/write head instead of ascending LBN order. Thus, all sectors on a track can be accessed in a single rotation, regardless of which sector passes under the head first.

Combined, these benefits can increase large access efficiency by 25-55%, depending on seek penalties. They can also make disk access times much more predictable, which is important to real-time applications (e.g., video

servers). However, these benefits are fully realized only for accesses that are track-sized and track-aligned. We believe that such accesses can be made much more common if file systems were modified to use *track-aligned extents*, which are extents specifically sized and aligned to match track boundaries.

4.2 Applications of Track-aligned Extents

Track-aligned extents are most valuable for workloads that involve many large requests to distinct portions of the disk. There are several important application workloads that fit this model. Here, we outline two:

Scanning of large files: Almost any application that scans large files will be well suited to track-aligned extents. This is true because most file systems allocate only a certain number of file blocks contiguously before starting a new contiguous region somewhere else. They do this to avoid exhausting the supply of unused space in any one part of the disk. The consequence, however, is that reading a large file does not result in streaming disk bandwidth — instead, it results in non-contiguous requests for relatively large amounts of data (e.g., 64KB or 256KB). We envision changing the allocations that determine these requests to utilize track-aligned extents.

Video servers: Video servers represent an ideal application of track-aligned extents. Although many people envision video as a "streaming media," this is not how the storage access patterns look in practice. Instead, relatively large segments of video data are read individually, at a rate that allows the video to be displayed smoothly. A video server will interleave the segment fetches for several videos such that they all keep up. The result is non-contiguous requests, where the discontinuities are due to timeliness requirements rather than allocation decisions. In addition, the number of videos that can be played simultaneously depends both on the average-case performance and the bounds that can be placed on response times. Thus, track-aligned extents can substantially increase video server throughput by making video segment fetches both more efficient and more predictable.

4.3 Realizing Track-aligned Extents

Using track-aligned extents in file systems (or other host software) requires two main changes:

1. The file system must discover the exact track boundaries. This is more complex than it might initially appear. Of course, zoned recording create a straightforward complexity in the mapping of LBNs to physical tracks. Much more irregularity in the mapping is caused by defect management, where unused spare regions and changes induced by defects can have widespread effects. To make matters worse, no two disk make/models have exactly the same mapping scheme, and no two disks have exactly the same mapping (because they generally have different defects).

2. The file system must change its internal algorithms to work with tracks, which have non-constant sizes and boundaries. For example, the allocation schemes and metadata structures must work with tracks. Also, fetch and write-back routines must be modified to extend accesses to needed blocks to the entire track that includes them.

4.4 Preliminary Progress

With simulations and analytic evaluation, we have convinced ourselves of the potential benefits of track-aligned extents. As discussed above, 25-55% increases in per-access efficiency can be achieved for "large" requests. We also have prior work on which to build in prototyping track-aligned extents; specifically, our DIXtrac tool [44] can automatically and efficiently extract the exact LBN-to-physical mapping from most SCSI disks (in addition to many other performance characteristics). Of course, we would prefer that the disk simply expose this information directly.

Much work remains in evaluating the costs and benefits of track-aligned extents in practice. Towards this end, we will experiment directly to identify appropriate implementation techniques and quantify the trade-offs involved. One of the interesting issues we've already run across is that the in-order bus delivery of data hides a fair portion of the benefits arising from out-of-order media access. Thus, another example of where storage interface change could be valuable is in allowing out-of-order bus delivery.

5. Example #2: Freeblock Scheduling

Disk firmware includes support for aggressive scheduling of media accesses in order to maximize efficiency. In its current form, however, this scheduling concerns itself only with overall throughput. As a result, host software does not entrust disk firmware with scheduling decisions for large sets of mixed-priority requests. Freeblock scheduling is a new approach to request scheduling that utilizes the accurate predictions needed for aggressive scheduling to combine the best of both worlds: minimized response times for high-priority requests with improved efficiency and steady forward progress for lower-priority requests. Specifically, freeblock scheduling replaces the rotational latency delays of high-priority requests with background media transfers. This section briefly describes freeblock scheduling, how it can be accomplished, application uses for it, and preliminary results indicating great promise.

5.1 Overview

Disk drives increasingly limit performance in many computer systems, creating complexity and restricting functionality. However, in recent years, the rate of improvement in media bandwidth (40%+ per year) has stayed close to that of computer system attributes that are driven by Moore's Law. It is only the mechanical positioning aspects (i.e., seek times and rotation speeds) that fail to keep pace. If 100% utilization of the potential media bandwidth could be realized, disk performance would scale roughly in proportion to the rest of the system over time. Unfortunately, utilizations of 2-15% are more commonly observed in practice.

Freeblock scheduling is a new approach to increasing media bandwidth utilization. By interleaving low priority disk activity with the normal workload (here referred to as background and foreground, respectively), one can replace many foreground rotational latency delays with useful %, though lower-priority, background media transfers. With appropriate freeblock scheduling, background tasks can receive 20-50% of a disk's potential media bandwidth without any increase in foreground request service times. Thus, this background disk activity is completed for free during the mechanical positioning for foreground requests.

There are many disk-intensive background tasks that are designed to occur during otherwise idle time. Examples include disk reorganization, file system cleaning, backup, prefetching, write-back, integrity checking, RAID scrubbing, virus detection, tamper detection, report generation, and index reorganization. When idle time does not present itself, these tasks either compete with foreground tasks or are simply not completed. Further, when they do compete with other tasks, these background tasks do not take full advantage of their relatively loose time constraints and paucity of sequencing requirements. As a result, these "idle time" tasks often cause performance or functionality problems in busy systems. With freeblock scheduling, background tasks can operate continuously and efficiently, even when they do not have the system to themselves.

5.2 Free Bandwidth and Freeblock Scheduling

At a high-level, the time required for a disk media access, T_{access}, can be computed.

$$T_{access} = T_{seek} + T_{rotate} + T_{transfer}$$

Of T_{access} only the $T_{transfer}$ component represents useful utilization of the disk head. Unfortunately, the other two components generally dominate. Many data placement and scheduling algorithms have been devised to increase disk head utilization by increasing transfer sizes and reducing positioning overheads. Freeblock scheduling complements these techniques by transferring additional data during the T_{rotate} component of T_{access}.

Fundamentally, the only time the disk head cannot be transferring data sectors to or from the media is during a seek. In fact, in most modern disk drives, the firmware will transfer a large request's data to or from the media "out of order" to minimize wasted time; this feature is sometimes referred to as zero-latency or immediate access. While seeks are unavoidable costs associated with accessing desired data locations, rotational latency is an artifact of not doing something more useful with the disk head. Since disk platters rotate constantly, a given sector will rotate past the disk head at a given time, independent of what the disk head is doing up until that time. So, there is an opportunity to do something more useful than just waiting for desired sectors to arrive at the disk head.

Freeblock scheduling consists of predicting how much rotational latency will occur before the next foreground media transfer, squeezing some additional media transfers into that time, and still getting to the destination track in time for the foreground transfer. The additional media transfers may be on the current or destination tracks, on another track near the two, or anywhere between them, as illustrated in Figure 3. In the two latter cases, additional seek overheads are incurred, reducing the actual time available for the additional media transfers, but not completely eliminating it.

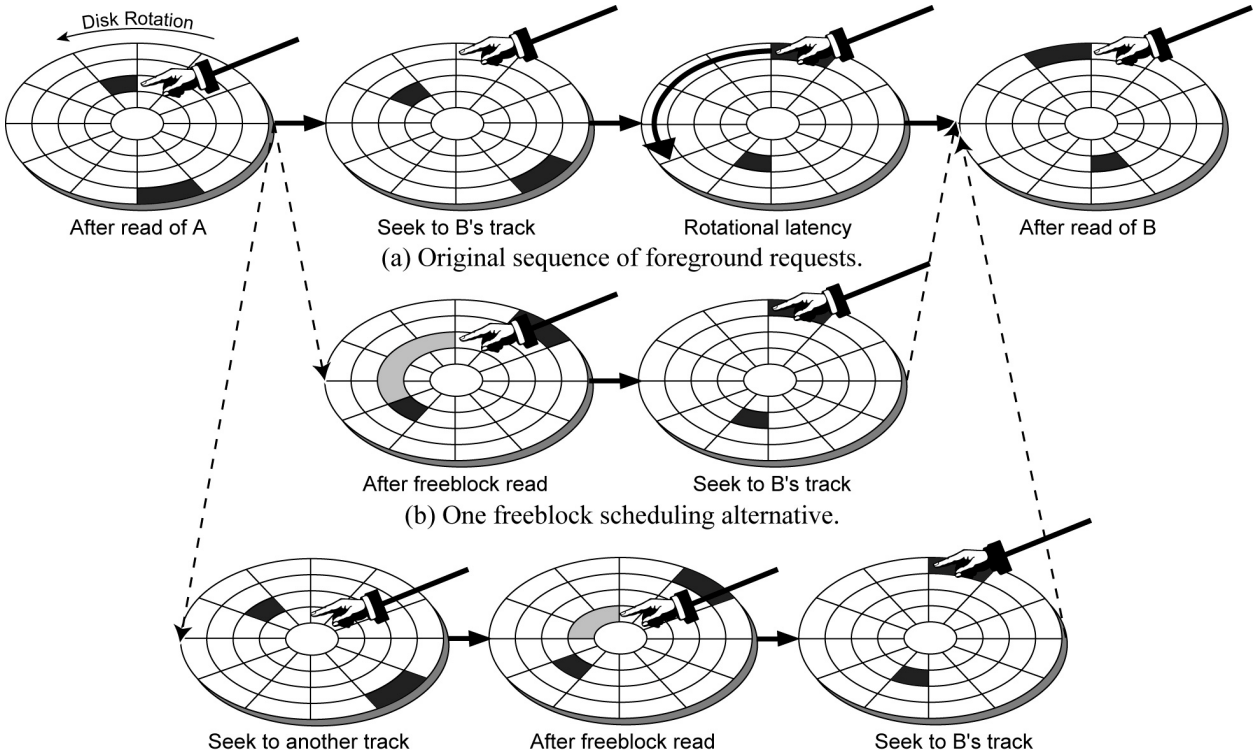

Figure 3: Illustration of two freeblock scheduling possibilities. Three sequences of steps are shown, each starting after completing the foreground request to block *A* and finishing after completing the foreground request to block *B*. Each step shows the position of the disk platter, the read/write head (shown by the pointer), and the two foreground requests (in black) after a partial rotation. The top row, labeled (a), shows the default sequence of disk head actions for servicing request *B*, which includes 4 sectors worth of potential free bandwidth (a.k.a. rotational latency). The second row, labeled (b), shows free reading of 4 blocks on *A*'s track using 100% of the potential free bandwidth. The third row, labeled (c), shows free reading of 3 blocks on another track, yielding 75% of the potential free bandwidth.

Accurately predicting future rotational latencies requires detailed knowledge of many disk performance attributes, including layout algorithms and time-dependent mechanical positioning overheads. These predictions can utilize the same basic algorithms and information that most modern disks employ for their internal scheduling decisions, which are based on overall positioning overheads (seek time plus rotational latency) [49, 24]. However, this may require that freeblock scheduling decisions be made by disk firmware. Fortunately, the increasing processing capabilities of disk drives [1, 18, 25, 41] make advanced on-drive storage management feasible [18, 54].

5.3 Applications of Free Bandwidth

Freeblock scheduling is a new tool, and we expect that system designers will find many unanticipated uses for it. This section describes some of the applications we see for its use.

Scanning applications. In many systems, there are a variety of support tasks that scan large portions of disk contents. Such activities are of direct benefit to users, although they may not be the highest priority of the system. Examples of such tasks include report generation, RAID scrubbing, virus detection, tamper detection [26], and backup. Our preliminary work has explored data mining of an active transaction processing system as a concrete example of such use of free bandwidth.

Internal storage optimization. Another promising use for free bandwidth is internal storage system optimization. Many techniques have been developed for reorganizing stored data to improve performance of future accesses. Examples include placing related data contiguously for sequential disk access [30, 54], placing hot data near the center of the disk [53, 43, 2], and replicating data on disk to provide quicker-to-access options for subsequent reads [34, 57]. Other examples include index reorganization [23, 20] and compression of cold data [9]. Our preliminary work has explored segment cleaning in log-structured file systems as a concrete example of such use of free bandwidth.

Prefetching and prewriting. Another use of free bandwidth is for anticipatory disk activities such as prefetching and prewriting. Prefetching is well-understood to offer significant performance enhancements [36, 6, 21, 28, 50]. Free bandwidth prefetching should increase performance further by avoiding interference with foreground requests and by minimizing the opportunity cost of aggressive predictions. As one example, the sequence shown in Figure 3(b) shows one way that the prefetching common in disk firmware could be extended with free bandwidth. Prewriting is the same concept in reverse. That is, prewriting is early writing out of dirty blocks under the assumption that they will not be overwritten or deleted before write-back is actually necessary. As with prefetching, the value of prewriting and its relationship with non-volatile memory are well-known [3, 8, 4, 20].

5.4 Preliminary Progress

In a recent paper [29], we described freeblock scheduling and quantified its potential given various disk and workload characteristics. By servicing background requests in the context of mechanical positioning for normal foreground requests, we have shown that 20-50% of a disk's potential media bandwidth is available with no impact on the original requests. Using simulation, the paper also demonstrates the value of freeblock scheduling with concrete examples of its use for storage system management and disk-intensive applications. The first example shows that cleaning in a log-structured file system can be done for free even when there is no truly idle time, resulting in up to a 300% speedup. The second example explores the use of free bandwidth for data mining on an active on-line transaction processing (OLTP) system, showing that over 47 full scans per day of a 9GB disk can be made with no impact on OLTP performance.

While freeblock scheduling can provide free media bandwidth, use of such bandwidth also requires some CPU, memory, and bus resources. One approach to addressing these needs is to augment disk drives with extra resources and extend disk firmware with application-specific functionality [1, 25, 41]. Potentially, such resources could turn free bandwidth into free functionality; in another recent paper [40], we argue exactly this case for the data mining example referred to above.

These results indicate significant promise, but additional experience is needed to refine and realize freeblock scheduling in practice. For example, it remains to be seen whether freeblock scheduling can be implemented outside of modern disk drives, given their high-level interfaces and complex firmware algorithms. Even inside disk firmware, freeblock scheduling will need to conservatively deal with seek and settle time variability, which may reduce its effectiveness. More importantly, new interfaces will be needed for hosts to expose freeblock requests to devices and to asynchronously receive the data. Exploring the issues involved with making freeblock scheduling a reality will be an important case study for our general research vision of more expressive interfaces.

6. Impact of Proposed Research

The ultimate impact of the proposed work will be to enable end-to-end specialization of storage functionality to application needs. By exploring the space of available device and workload information and different ways that it can be used, we hope to identify clear extensions to storage interfaces and algorithms, moving towards this goal.
This section outlines some additional views of this effort's potential impact.

Towards 100% utilization of media bandwidth. One of the most common complaints in computer systems is "disk performance isn't keeping up." To a certain extent, this is true. However, the engineers that make disk drives do amazing things, far exceeding the "wow" factor in most other aspects of computer science and engineering. In addition to amazing mechanical feats, their efforts the past few years are delivering 100%+ per year growth in density and ~40%+ per year growth in media bandwidth. If media bandwidth were fully utilized, disk performance would be keeping up. While it is unlikely that we will reach 100% utilization, harnessing more than the common 2-15% will be critical to continued evolution of storage and computer systems.

The value of simply starting to work together. Even ignoring the idyllic picture painted above, there is clear value to getting device firmware and host software to communicate and cooperate rather than each continuing to try to guess what the other is up to. The technologies on each side of the storage interface boundary have progressed beyond the other side's understanding. The result is frustration and stagnation. For example, disk firmware folks complain of short request queues, which frustrate them because they can do efficiency-scheduling better than host software. Similarly, file system people have given up on detailed data placement, focusing instead on just trying to use "large" requests to amortize positioning times over more data transfer.

References

[1] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: programming model, algorithms and evaluation. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3-7 October 1998), pages 81-91. ACM, 1998.

[2] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems,* 13 (2):89-121, May 1995.

[3] Mary Baker, Satoshi Asami, Etienne Deprit, John Ousterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. *Architectural Support for Programming Languages and Operating Systems* (Boston, MA, 12-15 October 1992). Published as Computer Architecture News, 20 (special issue): 10-22, 1992.

[4] Prabuddha Biswas, K. K. Ramakrishnan, and Don Towsley. Trace driven analysis of write caching policies for disks. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 13-23, 1993.

[5] Angela Demke Brown and Todd C. Mowry. Taming the memory hogs: using compiler-inserted releases to manage physical memory intelligently. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23-25 October 2000), pages 31-44. USENIX Association, 2000.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14 (4): 311-343, November 1996.

[7] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. The TickerTAIP parallel RAID architecture. *ACM International Symposium on Computer Architecture* (San Diego, CA), pages 52-63, 16-19 May 1993.

[8] Scott C. Carson and Sanjeev Setia. Analysis of the periodic update write policy for disk cache. *IEEE Transactions on Software Engineering*, 18 (1): 44-54, January 1992.

[9] Vincent Cate and Thomas Gross. Combining the concepts of compression and caching for a two-level filesystem. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, CA 8-11 April 1991), pages 200-211. ACM, 1991.

[10] Fay Chang and Garth A. Gibson. Automatic I/O hint generation through speculative execution. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22-25 February 1999), pages 1-14. ACM, Winter 1998.

[11] Yuqun Chen, Angelos Bilas, Stefanos N. Damianakis, Cezary Dubnicki, and Kai Li. UTLB: a mechanism for address translation on network interfaces. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3-7 October 1998). Published as *SIGPLAN Notices*, 33 (11): 193-204, November 1998.

[12] Tzi cker Chiueh. Trail: a track-based logging disk architecture for zero-overhead writes. Computer Science Department, State University of New York at Stony Brook, 1 October 1993.

[13] Peter J. Denning. Effects of scheduling on file memory operations. *AFIPS Spring Joint Computer Conference* (Atlantic City, New Jersey, 18-20 April 1967), pages 9-21, April 1967.

[14] Robert M. English and Alexander A. Stepanov. Loge: a self-organizing storage device. *Winter USENIX Technical Conference* (San Francisco, CA), pages 237-251. Usenix, 20-24 January 1992.

[15] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. *Annual USENIX Technical Conference* (Anaheim, CA), pages 1-17, January 1997.

[16] Gregory R. Ganger and Yale N. Patt. Using system-level models to evaluate I/O subsystem designs. *IEEE Transactions on Computers*, 47 (6): 667-678, June 1998.

[17] Gregory R. Ganger, Bruce L. Worthington, Robert Y. Hou, and Yale N. Patt. Disk arrays: high-performance, high-reliability storage systems. *IEEE Computer*, 27 (3): 30-36, March 1994.

[18] Garth A. Gibson, David F. Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. *Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, 3-7 October 1998). Published as SIGPLAN Notices, 33 (11): 92-103, November 1998.

[19] Garth A. Gibson, David F. Nagle, Khalil Amiri, Fay A. Chang, Howard Gobioff, Erik Riedel, David Rochberg,and Jim Zelenka. *Filesystems for network-attached secure disks*. CMU-CS-97-118. Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, July 1997.

[20] Richard Golding, Peter Bosch, Carl Staelin, Tim Sullivan, and John Wilkes. Idleness is not sloth. *Winter USENIX Technical Conference* (New Orleans, LA, 16-20 January 1995), pages 201-212. USENIX Association, 1995.

[21] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. *Summer USENIX Technical Conference* (Boston, MA, June 1994), pages 197-207. USENIX Association, 1994.

[22] Lan Huang and Tzi cker Chiueh. *Implementation of rotation latency sensitive disk scheduler*. Technical report ECSL TR-81. SUNY Stony Brook, December 1999.

[23] Erin H. Herrin II and Raphael Finkel. An ASCII database for fast queries of relatively stable data. *Computing Systems*, 4 (2): 127-155. Usenix Association, Spring 1991.

[24] David M. Jacobson and John Wilkes. *Disk scheduling algorithms based on rotational position*. Technical report HPL-CSP-91-7. Hewlett-Packard Laboratories, Palo Alto, CA, 24 February 1991, revised 1 March 1991.

[25] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (IDISKs). *SIGMOD Record*, 27 (3): 42-52, September 1998.

[26] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: a file system integrity checker. *Conference on Computer and Communications Security* (Fairfax, VA, 2-4 November 1994), pages 18-29, 1994.

[27] David Kotz. Disk-directed I/O for MIMD multiprocessors. *Symposium on Operating Systems Design and Implementation* (Monterey, CA), pages 61-74. Usenix Association, 14-17 November 1994.

[28] Thomas M. Kroeger and Darrell D. E. Long. The case for efficient file access pattern modeling. *Hot Topics in Operating Systems* (Rio Rico, Arizona, 29-30 March 1999), pages 14-19, 1999.

[29] Christopher R. Lumb, Jiri Schindler, Gregory R. Ganger, David F. Nagle, and Erik Riedel. Towards higher disk head utilization: extracting free bandwidth from busy disk drives. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23-25 October 2000), pages 87-102. USENIX Association, 2000.

[30] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5-8 October 1997). Published as Operating Systems Review, 31 (5): 238- 252. ACM, 1997.

[31] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2 (3): 181-197, August 1984.

[32] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. *Winter USENIX Technical Conference* (Dallas, TX), pages 33-43, 21-25 January 1991.

[33] Rodney Van Meter. Observing the effects of multi-zone disks. *Annual USENIX Technical Conference* (Anaheim, CA), pages 19-30, 6-10 January 1997.

[34] Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, 40 (1): 22- 30, January 1991.

[35] *Object based storage devices: a command set proposal*. Technical report. October 1999. http://www.T10.org/.

[36] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. *ACM Symposium on Operating System Principles* (Copper Mountain Resort, CO, 3-6 December 1995). Published as Operating Systems Review, 29 (5): 79-95, 1995.

[37] Quantum Corporation. *Quantum Viking 2.27/4.55 GB S product manual*, Document number 81-113146-02, April 1997.

[38] Quantum Corporation. *Quantum Atlas 10K 9.1/18.2/36.4 GB SCSI product manual*, Document number 81-119313-05, August 1999.

[39] *ReiserFS*. Technical report

[40] Erik Riedel, Christos Faloutsos, Gregory R. Ganger, and David F. Nagle. Data mining on an OLTP system (nearly) for free. *ACM SIGMOD International Conference on Management of Data* (Dallas, TX, 14-19 May 2000), pages 13-21, 2000.

[41] Erik Riedel, Garth Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia applications. *International Conference on Very Large Databases* (New York, NY, 24-27 August, 1998). Published as Proceedings VLDB., pages 62-73. Morgan Kaufmann Publishers Inc., 1998.

[42] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10 (1): 26-52, February 1992.

[43] Chris Ruemmler and John Wilkes. *Disk Shuffling*. Technical report HPL-91-156. Hewlett-Packard Company, Palo Alto, CA, October 1991.

[44] Jiri Schindler and Gregory R. Ganger. *Automated disk drive characterization*. Technical report CMU-CS-99-176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[45] Seagate. *Seagate Cheetah 4 LP Family: ST34501N/W/WC/WD/DC product manual*, volume 1, Document number 83329120, April 1997.

[46] Seagate. *Seagate Cheetah X15 FC disk drive ST318451FC/FCV product manual, volume 1*, Document number 83329486, June 2000.

[47] P. H. Seaman, R. A. Lind, and T. L. Wilson. On teleprocessing system design, part IV: an analysis of auxiliary-storage activity. *IBM Systems Journal*, 5 (3): 158-170, 1966.

[48] Margo Seltzer. *A comparison of data manager and file system supported transaction processing*. Computer Science Div., Department of Electrical Engineering and Computer Science, University of California at Berkeley, January 1990.

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

[50] Liddy Shriver, Keith Smith, and Chris Small. Why does filesystem prefetching work? *USENIX* (Monterey, California), pages 71-83, 6-11 June 1999.

[51] David C. Steere. Exploiting the non-determinism and asynchrony of set iterators to reduce aggregate file I/O latency. *ACM Symposium on Operating System Principles* (Saint-Malo, France, 5-8 October 1997). Published as Operating Systems Review, 31 (5): 252-263. ACM, 1997.

[52] Manolis M. Tsangaris and Jeffery F. Naughton. On the performance of object clustering techniques. *ACM SIGMOD International Conference on Management of Data* (San Diego, California), pages 144-153, 2-5 June 1992.

[53] Paul Vongsathorn and Scott D. Carson. A system for adaptive disk rearrangement. *Software: Practice and Experience*, 20 (3): 225-242, March 1990.

[54] Randolph Y. Wang, Thomas E. Anderson, and Michael D. Dahlin. *Experience with a distributed file system implementation*. Technical Report CSD-98-986. University of California at Berkeley, January 1998.

[55] Bruce L. Worthington, Gregory R. Ganger, and Yale N. Patt. *Scheduling for modern disk drives and non-random workloads*. CSE-TR-194-94. Department of Computer Science and Engineering, University of Michigan, March 1994.

[56] Bruce L. Worthington, Gregory R. Ganger, Yale N. Patt, and John Wilkes. On-line extraction of SCSI disk drive parameters. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Ottawa, Canada), pages 146-156, May 1995.

[57] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23-25 October 2000), pages 243-258. USENIX Association, 2000.