Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics

Jiri Schindler, John Linwood Griffin, Christopher R. Lumb,
Gregory R. Ganger

April 2001
CMU-CS-01-119

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Track-aligned extents (traxtents) utilize disk-specific knowledge to match access patterns to the strengths of modern disks. By allocating and accessing related data on disk track boundaries, a system can avoid most rotational latency and track crossing overheads. Avoiding these overheads can increase disk access efficiency by up to 50% for mid-sized requests (100–500 KB). This paper describes traxtents, algorithms for detecting track boundaries, and use of traxtents in file systems and video servers. For large file workloads, a modified version of FreeBSD's FFS implementation reduces application run times by 20% compared to the original version. A video server using traxtent-based requests can support 56% more concurrent streams at the same startup latency and buffer space. For LFS, 44% lower overall write cost for track-sized segments can be achieved.

We thank the members and companies of the Parallel Data Consortium (at the time of this writing: EMC Corporation, Hewlett-Packard Labs, Hitachi, IBM Corporation, Intel Corporation, LSI Logic, Lucent Technologies, Network Appliances, PANASAS, Platys Communications, Seagate Technology, Snap Appliances, Sun Microsystems, Veritas Software Corporation) for their insights and support. This work is partially supported by the National Science Foundation via CMU's Data Storage Systems Center. John Griffin is supported in part by a National Science Foundation Graduate Fellowship.

Keywords: data layout, disk characterization, disk efficiency, file systems, track-aligned access

1 Introduction

Rotating media has come full circle, so to speak. The first uses of disks in the 1950s ignored the effects of geometry in the interest of getting a working system. Later, algorithms were developed that paid attention to disk geometry in order to improve disk efficiency. These algorithms were often hard-coded and hardware-specific, making them fragile across generations of hardware. To address this, a layer of abstraction was standardized between operating systems and disks, virtualizing disk storage as a flat array of fixed-sized blocks. Unfortunately, this abstraction hides too much information, making the OS's task of maximizing disk efficiency more difficult than necessary.

File systems and databases attempt to mitigate the ever-present disk performance problem by aggressively clustering on-disk data and issuing fewer, larger disk requests. This is done with only a vague understanding of disk characteristics, focusing on the notion that bigger requests are better because they amortize positioning delays over more data transfer. Although this notion is generally correct, there are performance and complexity costs associated with making requests larger and larger. For video servers, ever-larger request sizes result in an increase in both buffer space requirements and stream initiation latency [6, 7, 14, 17, 24]. Log-structured file systems (LFS) incur a greater cleaning overhead as segment size increases [5, 18, 27]. Even for general file system operation, allocation of very large sequential regions competes with space management robustness [19], and very large accesses may put deep prefetching ahead of foreground requests. In some circumstances, useful large requests are not even possible; for example, many files are too small. These examples all indicate that achieving higher disk efficiency with smaller request sizes would be valuable.

This paper describes and analyzes track-aligned extents (*traxtents*), extents that are aligned and sized so as to match the corresponding disk track size. By exploiting a small amount of disk-specific knowledge in this way, a system can significantly increase the efficiency of mid-to-large requests (100 KB and up). Traxtent-aware access yields up to 50% higher disk efficiency. This improvement stems from two main sources. First, track-aligned access minimizes the number of track switch delays, which have not improved much over the years and are now significant (0.6–1.1 ms) relative to other delays. Second, full-track access eliminates rotational latency (3 ms per request on average at 10,000 RPM) for disk drives whose firmware supports zero-latency access. Point A of Figure 1 shows random track-aligned accesses yielding an efficiency within 82% of the maximum possible, whereas unaligned accesses only achieve 56% of the best-case for the same request size.

Using traxtents requires three main system software changes. First, the track boundaries must be identified. This task is more difficult than might be expected, because of the zoned recording and media defect management common in modern disk drives. Second, disk space allocation and placement algorithms must be changed to situate data in traxtents. Third, request-generating routines must be changed to utilize traxtents as the unit of access. Since different traxtents will have different sizes due to zone boundaries and media defects, this variability must be handled by the system software. Supporting this variability at the system level is also sufficient to avoid hardware-specific dependencies. This paper discusses these system-level changes and how they are handled in a prototype implementation of a traxtent-aware FFS file system in FreeBSD 4.0. This implementation includes new algorithms for identifying track boundaries.

We evaluate track-based access with both detailed disk measurements and overall system performance measurements. The former shows increased disk efficiency, reduced access time variance, and system requirements that must be satisfied to achieve the highest efficiency. The latter show promising improvements in several situations. For example, when accessing two large files concurrently, the traxtent-aware FFS yields 20% higher performance compared to current defaults. In exploring video server workloads, we observe an ability to support either 56% more concurrent streams at the same startup latency or a 5× reduction in startup latency and buffer space for the maximum number of streams supported by the video server. Finally, we compute 44% lower overall write cost for track-sized segments in LFS.

The remainder of this paper is organized as follows. Section 2 motivates track-based access by describing the technology drivers and expected benefits in more detail. Section 3 describes system changes required for traxtents. Section 4 describes our implementation of traxtents in FreeBSD. Section 5 evaluates traxtents under a variety of circumstances. Section 6 discusses related work. Section 7 summarizes this paper's contributions.

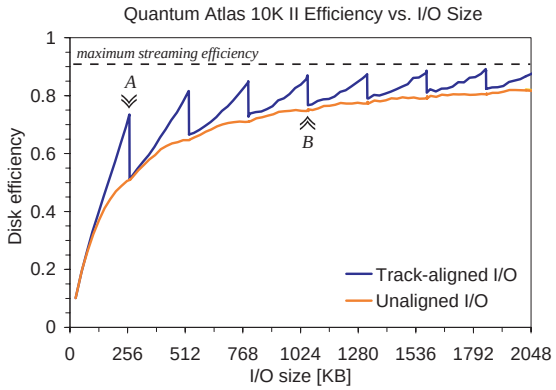


Figure 1: **Measured advantage of track-aligned access over unaligned access.** This graph plots disk efficiency as a function of I/O size. We define disk efficiency as a fraction of total access time (which includes seek and rotational latency) spent reading or writing data to the media. The access time was measured on a random read workload to a Quantum Atlas 10K II disk's first zone of size 264 KB. The maximum achievable streaming efficiency is less than 1 because no data is transferred during a head switch. Point A highlights the higher efficiency of track-aligned access (0.73, or 82% of the maximum possible) over unaligned access for a track-sized request. Point B shows where unaligned I/O efficiency catches up to the aligned efficiency at Point A. The drop-offs in the track-aligned curve occurs whenever the aligned access crosses a track boundary. Note that the highest efficiency is achieved whenever I/O size is a multiple of the track size.

2 Track-based Disk Access

In determining what data to read and write and when, system software attempts to maximize overall performance in the face of two competing pressures. On the one hand, the underlying disk technology pushes for larger request sizes in order to maximize disk efficiency. Specifically, time-consuming mechanical delays can be amortized by transferring large amounts of data between each repositioning of the disk head. For example, Point B of Figure 1 shows that reading or writing 1 MB at a time results in a 75% disk efficiency for normal (track-unaligned) access. On the other hand, complexity and other resource limitations can impose a cost on the use of very large requests. For example, buffer space limitations and irregular application behavior may prevent the higher disk efficiency of larger requests from translating into improved overall performance. This section discusses the system pressures that push for smaller request sizes, the disk characteristics that make track-based accesses particularly efficient, and the types of applications that will benefit most from track-based disk access.

2.1 Limitations on request size

System software designers would like to be able to always use the large disk requests that maximize efficiency. Unfortunately, in practice, resource limitations and imperfect information about future accesses make this difficult. Pressure against ever-larger requests occurs for four different reasons: (1) responsiveness, (2) limited buffer space, (3) irregular access patterns, and (4) storage space management.

Responsiveness. Although larger requests increase disk efficiency, they do so at the expense of higher latency. This trade-off between efficiency and responsiveness is a recurring theme in computer systems, and it is particularly steep for disk systems. The latency increase can manifest itself in several different ways. At the local level, the non-preemptive nature of disk requests combined with the long access times of large requests (35–50 ms for 1 MB requests) result in substantial I/O wait times for small, synchronous requests. This problem has been noted for both FFS and LFS [5, 31]. At the global level, grouping substantial quantities of data into large disk writes usually requires heavy use of write-back caching. Although application performance is usually decoupled from the eventual write-back, application changes are not persistent until the disk writes complete. Making matters worse, the amount of data that must be delayed and buffered to achieve large enough writes continues to grow. As another example, many video servers schedule

fetches of video segments in carefully-determined rounds of disk requests. Using larger disk requests increases the time for each round, which increases the time required to start streaming a new video. Section 5.5 quantifies the start-up latency required for modern disks.

Buffer Space. Although memory sizes continue to grow, they remain finite. Larger disk requests stress memory resources in two ways. For reads, larger disk requests are usually created by fetching more data farther in advance of the actual need for it; this prefetched data must be buffered until it is needed. For writes, larger disk requests are usually created by holding more data in write-back cache until enough contiguous data is dirty; this dirty data must be buffered until it is written to disk. The persistence problem discussed above can be addressed with non-volatile RAM buffers, but the buffer space issue will remain.

Irregular Access Patterns. Large disk requests are most easily generated when applications use regular access patterns and large files. Although sequential full-file access is relatively common [23, 1, 38], most data objects are much smaller than the disk request sizes needed to achieve good disk efficiency. For example, most files are well below 32 KB in size in UNIX-like systems [12, 33] and below 64 KB in Microsoft Windows systems [38, 10]. Directories and file attribute structures are almost always much smaller. To achieve sufficiently large disk requests in such environments, access patterns across data objects must be predicted at on-disk layout time. Although approaches to grouping small data objects have been explored [27, 12, 15, 26, 11], all are based on imperfect heuristics, and thus they rarely group things perfectly. Even though disk efficiency is higher, misgrouped data objects result in wasted disk bandwidth and buffer memory, since some fetched objects will go unused. Further, as the target request sizes grow, identifying sufficiently strong inter-relationships becomes more difficult.

Storage Space Management. Large disk requests are only possible when closely related data is collocated on the disk. Achieving this collocation requires that on-disk placement algorithms be able to find large regions of free space when needed. Also, when grouping multiple data objects together, growth of individual data objects must be accommodated. All of these needs must be met with little or no information about the sequence of future storage allocation and deallocation operations. Collectively, these facts create a complex storage management problem. Systems can address this problem with combinations of pre-allocation heuristics [4, 16], on-line reallocation actions [27, 34, 35], and idle-time reorganization [18, 2]. There is no straightforward solution, and the difficulty grows with the target disk request size.

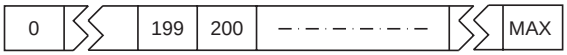
2.2 Disk characteristics

Modern storage protocols, such as SCSI and IDE/ATA, expose storage capacity as a linear array of fixed-sized blocks (Figure 2(a)). By building atop this abstraction, OS software need not concern itself with complex device-specific details, and code can be reused across the large set of storage devices that use these interfaces (e.g., disk drives and disk arrays). Likewise, by exposing only this abstract interface, storage device vendors are free to modify and enhance their internal implementations. Behind this interface, the storage device must translate the logical block numbers (LBNs) to physical storage locations. Figure 2(b) illustrates this translation for a disk drive, wherein LBNs are assigned sequentially on each track before moving to the next. Disk drive advances over the past decade have conspired to make the track a sweet-spot for disk efficiency, yielding the 50% increase at Point A of Figure 1. This section describes these advances.

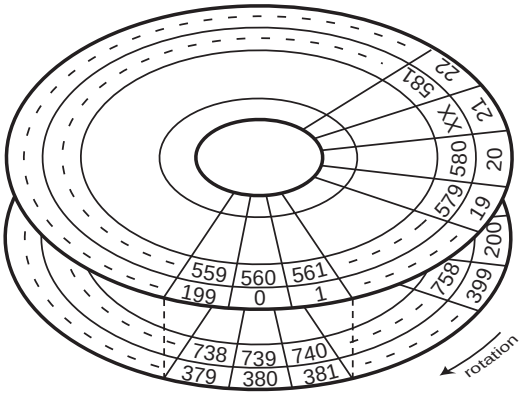
Head Switch. A head switch occurs when a single request accesses a sequence of LBNs whose on-disk locations span two different tracks. This head switch consists of turning on the electronics for the appropriate read/write head and adjusting its position. The position adjustment is necessary because the second track, which usually lies on a different surface, may not be perfectly aligned with the first one. Therefore, the disk has to read servo information to determine where the head is located and then shift the set of arms to center the head above the second track. The same actions are needed for single-track seeks on the same surface. In the example of Figure 2(b), head switches occur between logical blocks 199 and 200, 399 and 400, and 598 and 599.

Even compared to other disk characteristics, head switch time has improved only a little in the past decade. While disk rotation speeds have improved by a factor of 3× and average seek times by a factor of 2.5×, head switch times have decreased by only 20–40% (see Table 1). At 0.6–1.1 ms, the head switch time now takes about 1/5 of a revolution for a 15,000 RPM disk. These trends have increased the significance of head switch time.

Naturally, not all requests span track boundaries. The probability of a head switch, P_{hs}, depends on



(a) **System's view of storage.**



(b) **Mapping of LBNs onto physical sectors.**

Figure 2: **Standard system view of disk storage and its mapping onto physical disk sectors.** (a) illustrates the linear sequence of logical blocks, often 512 bytes, that the standard disk protocols expose. (b) shows one example mapping of those logical block numbers (LBNs) onto the disk media. The depicted disk drive has 200 sectors per track, two media surfaces, and track skew of 20 sectors. Logical blocks are assigned to the outer track of the first surface, the outer track of the second surface, the second track of the first surface, and so on. The track skew accounts for the head switch delay and ensures optimal streaming bandwidth. The picture also shows a defect between the sectors with LBN 580 and 581 which has been handled by slipping. Therefore, the first LBN on the following track is 599 instead of 600.

Disk	Year	RPM	Head Switch	Avg. Seek	Sectors per Track
HP C2247	1992	5400	1 ms	10 ms	96–56
Quantum Viking	1997	7200	1 ms	8.0 ms	216–126
IBM Ultrastar 18 ES	1998	7200	1.1 ms	7.6 ms	390–247
IBM Ultrastar 18LZX	1999	10000	0.8 ms	5.9 ms	382–195
Quantum Atlas 10K	1999	10000	0.8 ms	5.0 ms	334–224
Seagate Cheetah X15	2000	15000	0.8 ms	3.9 ms	386–286
Quantum Atlas 10K II	2000	10000	0.6 ms	4.7 ms	528–353

Table 1: **Representative disk characteristics.** Note the relatively small change in the head switch penalty compared with other disk improvements.

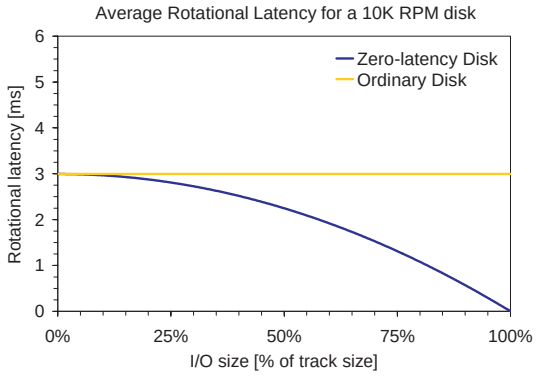


Figure 3: **Average rotational latency for ordinary and zero-latency disks as a function of request size.** The request size is expressed as a percentage of the track size. The rotational latency is shown in absolute terms for a 10,000 RPM disk.

workload and disk characteristics. For a request of S sectors and a track size of N sectors, $P_{hs} = (S-1)/N$, assuming that the requested locations are uncorrelated with track boundaries. With S approaching N, almost every request will involve a head switch, which is why we refer to conventional systems as "track-unaligned" even though they are only track-unaware. In this situation ($S = N$), track-aligned access improves the response time of most requests by the 0.6–1.1 ms head switch time. Even for smaller accesses, head switches can be common without track-alignment. For example, with 64 KB requests ($S = 128$) and an average track size of 192 KB ($N = 384$), a head switch occurs for every third access.

Zero-latency Access. The second modern disk feature that pushes for track-based access is zero-latency access, also known as immediate access or access-on-arrival. When a disk wants to read S contiguous sectors, the simplest approach is to position the head (by a combination of seek and rotational latency) to the first sector and read the S sectors in ascending LBN order. With zero-latency access support, a disk can read the S sectors from the media into its buffers in any order. In the best case of reading exactly one track, the head can start reading data as soon as the seek is completed; no rotational latency is required in this case because all sectors on the track are needed. The S sectors are read into an intermediate buffer, assembled in ascending LBN order, and sent to the host. The same concept applies to writes, except that data must be delivered from the host to the disk's buffers before being written onto the media.

As an example of zero-latency access on the disk from Figure 2(b), consider a read request to logical blocks 200–399. First, the head is moved to the track containing these blocks. Suppose that after the seek, the disk head is positioned above the sector containing LBN 380. A zero-latency disk can start reading immediately the contiguous range 380–399. Since all of the desired logical blocks are mapped onto a single track, the head keeps reading past the sector with LBN 399 and reads the remaining sectors with LBN 200–379. Thus, the entire track is read in only one rotation even though the head arrived in the "middle" of the track.

The expected rotational latency for a zero-latency disk decreases as the request size, S, increases as shown in Figure 3. An ordinary disk, on the other hand, has an expected rotational latency of $(N-1)/2N$, or approximately 1/2 revolution, regardless of the request size. Therefore, a request for all N sectors on a track requires only one revolution after the seek for a zero-latency disk, and anywhere from one to two (average of 1.5) otherwise.

2.3 Putting it all together

For requests around the track size (100–500 KB), the potential benefit of track-based access is substantial. A track-unaligned access for N media sectors involves four delays: seek, rotational latency, N sectors worth of media transfer, and head switch. A N-sector track-aligned request eliminates the rotational latency and head switch delays. This reduces access times for modern disks by 3–4 ms out of 9–12 ms, resulting in a 50% increase in efficiency.

Of course, the real benefit provided by track-based access depends on the workload. For example, a

workload of random small requests, as characterizes transaction processing, will see minimal improvement because request sizes are too small. At the other end of the spectrum, applications involving large sequential I/O activity will also see little benefit, because positioning costs can be amortized over large transfers. Track-based access provides most benefit to applications with medium-sized I/Os that have imperfect locality. In Section 5, we explore several examples of such applications, including large file accesses that share the disk with other activity, video servers, and log-structured file systems.

3 Traxtent-aware System Design

Track-based disk access is possible with relatively minor changes to existing systems. This section discusses practical design considerations involved with these changes.

3.1 Extracting track boundaries

In order to use track boundary information, systems must first obtain a list of where those boundaries occur. Specifically, systems must know the range of LBNs that map onto each track. Under ideal circumstances, the disk would provide this information directly. However, since modern SCSI and IDE/ATA disks do not, the track boundaries must be determined experimentally.

Extracting track boundaries is made difficult by the internal space management algorithms employed by disk firmware. In particular, three aspects complicate the basic LBN-to-physical mapping pattern. First, because outer tracks have greater circumference than inner tracks, modern disks record more sectors on the outer tracks. Typically, the set of tracks is partitioned into 8–20 subsets (referred to as zones or bands) with a different number of sectors per track in each zone. Second, because some amount of defective media is expected, some fraction of the disk's sectors are set aside as spare space for defect management. This spare space disrupts the pattern even when there are no defects. Worse, there are a wide array of spare space schemes (e.g., spare sectors per track, spare sectors per cylinder, spare tracks per zone, spare space at the end of the disk, etc.); we have observed over 10 distinct schemes in different disk makes and models. Third, when defects exist, the default LBN-to-physical mapping is complicated by the disk's avoidance strategy for the defective regions. Defect avoidance is handled in one of two ways: *slipping*, wherein the LBN-to-physical mapping is modified to simply skip the defective sector, and *remapping*, wherein the LBN that would be located in a defective sector is instead located elsewhere leaving other mappings unchanged. Slipping is more efficient and more common, but it affects the mappings of subsequent LBNs.

Although track detection can be complex, this extraction need be performed only once. Track boundaries only change in use if new defects "grow" on the disk, which is rare after the first 48 hours of operation [25].

3.2 Allocation and access

Extent-based systems such as NTFS [22] and XFS [36] allocate disk space to files by specifying ranges of LBNs (extents) associated with each file. Block-based file systems such as Ext2 [4] and FFS [19] group LBNs into fixed-size allocation units (blocks) typically 4 or 8 KB in size. Extent-based systems lend themselves naturally to track-based alignment of data: extent ranges during allocation can be chosen on the basis of track boundaries. Block-based systems can approximate track-sized extents by placing sequential runs of blocks such that they never span track boundaries. This wastes some space when track sizes are not evenly divisible by the block size. However, this space is usually less than 5% of total storage space and could be reclaimed by the system for storing inodes, superblocks, or fragmented blocks.

Once the system determines that a large file is being written, it may be useful to reserve (preallocate) entire traxtents even when writing less than a traxtent worth of data. It may also be useful to reserve traxtents for use by files that tend to be accessed as a group [12, 26] such that many small files can be read or written with a single track-sized request. When the file system becomes aged and fragmented it may be beneficial to support relocation of small files and fragments to free track-sized extents, which would be a process much like cleaning in a log-structured file system [27]. A similar relocation could be done for large files to retrofit existing disk partitions for traxtent-optimized access.

After allocation routines are modified to situate data on track boundaries, system software must also be extended to generate traxtent requests whenever possible. Usually, this will involve extending or clipping prefetch and write-back requests based on track boundaries.

Our experimentation has uncovered an additional design consideration: current systems will only realize the full benefit of traxtent-based requests when utilizing command queueing at the disk. Although zero-latency disks can access LBNs from the media in any order, the current SCSI and IDE/ATA protocols only allow for in-order delivery of data to/from the host. As a result, bus transfer overheads hide some of the benefit of zero-latency access. By having multiple requests outstanding at the disk, the next request's seek can be overlapped with the current request's bus transfer, yielding the full disk efficiency benefits shown in Figure 1. Fortunately, most modern disks and most current OSes support command queueing at the disk.

4 Implementation

We have developed a prototype implementation of a traxtent-aware file system in FreeBSD. This implementation identifies track boundaries and modifies the FreeBSD FFS implementation to take advantage of this information. This section describes our algorithms for detecting track boundaries and details our modifications to FFS.

4.1 Detecting track boundaries

We have implemented two approaches to detecting track boundaries: a general approach applicable to any disk interface supporting a read command and a specialized approach for SCSI disks.

4.1.1 General approach

Our extraction algorithm locates track boundaries by identifying discontinuities in access efficiency. Recall from Figure 1 that disk efficiency for track-aligned requests increases linearly with the number of sectors being transferred until a head switch occurs. Starting from sector with LBN 0 of the disk ($S = 0$), our algorithm issues successive requests of increasing size, each starting at sector S (i.e., read 1 sector starting at S, read 2 sectors starting at S, etc.). Eventually an N-sector read returns in more time than a linear model suggests, which identifies sector $S+N$ as the start of a new track. The process begins anew by setting $S = S + N$ and repeating the algorithm.

The method described above is clearly suboptimal; our actual implementation uses a binary search algorithm to find N. In addition, once a track size is determined, the common case of each subsequent track being the same size is quickly verified for the last found value of N. This verification checks if a discontinuity between $S + N - 1$ and $S + N$ occurs. If so, we set $S = S + N$ and repeat the verification, otherwise we set $N = 1$ and start anew from location S. This verification thus results in performing full extraction only on first tracks of a new zone or tracks containing defects. Using this technique, the track boundaries of a 9 GB disk (the Atlas 10K) are extracted in four hours. Talagala et al. describe a much quicker algorithm that extracts approximate geometry information using just the read command [37]; however, for our purposes the exact track boundaries must be identified.

A problem with using read requests to detect track boundaries is the sector prefetching and caching performed by disk firmware. To obviate the effects of firmware caching, we issue 100 parallel extraction operations to widespread locations such that the cache is flushed each time we return to block S. An alternative approach would be to use write operations; however, this is undesirable because of the destructive nature of writes and the fact that some disks implement delayed writes internally.

4.1.2 SCSI-specific approach

The SCSI command set supports query operations that can help with track boundary detection. Worthington et al. describe how these operations can be used to determine LBN-to-physical mappings [40]. Building upon their basic mechanisms, we have implemented an automated disk drive characterization tool called DIXtrac [29]. This tool includes a five-step algorithm that exploits the regularity of geometry and layout

characteristics to efficiently and automatically extract the complete LBN-to-physical mappings in less than one minute (fewer than 30,000 LBN translations), independent of disk capacity:

1. Use the READ CAPACITY command to determine the highest LBN, and determine the number of cylinders and surfaces by mapping random and targeted LBNs to physical locations using the SEND/RECEIVE DIAGNOSTIC command.
2. Use the READ DEFECT LIST command to obtain a list of all media defect locations.
3. Determine where spare sectors are located on each track and cylinder, and detect any other space reserved by the firmware. This is done by an expert-system-like process of combining the results of several queries, including whether or not (a) each track in a cylinder has the same number of LBN-holding sectors; (b) one cylinder within a set has fewer sectors than can be explained by the defect list; and (c) the last cylinder in a zone has too few sectors.
4. Determine zone boundaries and the number of sectors per track in each zone by counting the sectors on a defect-free, spare-free track in each zone.
5. Identify the remapping mechanism used for each defective sector. This is determined by back-translating the LBNs returned in step 2.

These steps are described in more detail in [29]. DIXtrac has been successfully used on 11 disk models from 4 different manufacturers.

4.2 Traxtent support in FreeBSD

This section reviews the basic operation of FreeBSD FFS [19] and describes our changes to implement traxtent-aware allocation and access in FreeBSD.

4.2.1 FreeBSD FFS overview

FreeBSD assigns three identifying block numbers to buffered disk data (Figure 4). The `lblkno` represents the offset within a file; that is, the buffer containing the first byte of file data is identified by `lblkno` 0. Each `lblkno` is associated with one `blkno` (physical block number), which is an abstract representation of the disk media used by the OS to simplify space management. Each `blkno` directly maps to a range of contiguous disk *sector numbers* (LBNs), which are the actual addresses presented to the disk drive during an access. In our experiments, the minimum disk request size is one physical block, which is 8 KB (sixteen contiguous LBNs). In this section, "block" refers to a physical block.

FFS clusters large sequential groups of disk sectors into fixed-size block groups ("cylinder groups"). Each block group contains a small amount of summary information—inodes, free block map, etc.—followed by a large contiguous array of data blocks. Block group size, block allocation and media access characteristics were once based on the underlying disk's physical geometry. Although this geometric dependence is no longer true, block groups are still used in their original form because they localize related data (e.g., files in the same directory) and their inodes, resulting in more efficient disk access. The block groups created for our experiments are 32 MB in size.

FFS uses the clustered allocation and access algorithms described by McVoy & Kleiman [20]. When newly created data are committed to disk, blocks are allocated to a file by selecting the closest "cluster" of free blocks (relative to the last block committed) large enough to store all N blocks of buffered data. In the common case (when the block group is sparsely populated with minimal fragmentation) the cluster selected contains the N blocks immediately following the last block committed. To ensure fair local allocation among multiple files, the system allows only half of the blocks in a block group to be allocated to a single file before switching to a new block group.

FFS implements a history-based readahead (prefetching) algorithm when reading large files sequentially. The system maintains a "sequential count" of the last run of sequentially accessed blocks (if the last four accesses were for blocks 17, 20, 21, and 22, the sequential count is 3). When the number of cached readahead blocks drops below 32, FFS issues a new readahead of length l beginning with the first noncached block,

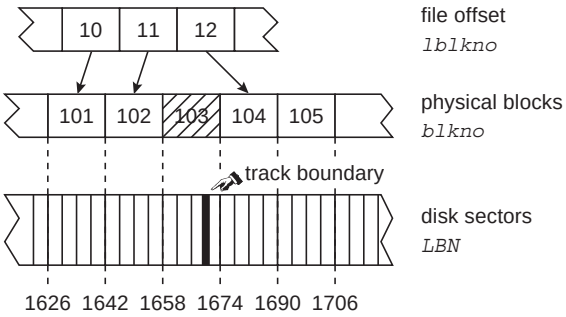


Figure 4: **Mapping system-level blocks to disk sectors.** Physical block 101 maps directly to disk sectors 1626–1641. Block 103 is an *excluded* block (see Section 4.2.2) because it spans the disk track boundary between LBNs 1669–1670.

where l is the lowest of (a) the sequential count, (b) the number of contiguously allocated blocks remaining in the current cluster, or (c) 32 blocks[1].

4.2.2 FreeBSD FFS modifications

Excluded blocks and traxtent allocation. We introduce the concept of the *excluded* block, highlighted in Figure 4. Our modified cluster allocation algorithm ignores excluded blocks when considering layout of large file data: whenever the preferred block (the next sequential block) is excluded, we instead allocate the first block of the closest available traxtent. When possible, mid-size files are allocated such that they fit within a single traxtent. On average, one out of every twenty blocks of the Quantum Atlas 10K is excluded under our modified FFS.

Traxtent-sized access. No fundamental changes are necessary in the FFS clustered readahead algorithm. FFS properly identifies runs of blocks between excluded blocks as clusters and accesses them with a single disk request. We eliminate the sequential access counter to prevent multiple partial accesses to a single traxtent. We handle the special case where there is no excluded block between traxtents by ensuring that no readahead request goes beyond a track boundary. At a low level, unmodified FreeBSD already supports command queuing at the device and attempts to have at least one outstanding request for each active data stream.

Traxtent data structures. When the file system is created, track boundaries are stored on disk. At mount time, they are read into an extended FreeBSD `mount` structure. We chose the mount structure because it is available everywhere traxtent information is needed. This approach also allows the use of traxtents to be controlled at mount time.

5 Evaluating Traxtents

We examine the performance benefits of track-based access at two levels. We begin by evaluating the disk in isolation, finding a 50% improvement in disk efficiency and a reduction in response time variance. We then quantify system-level performance gains, noting a 20% reduction in run time for large file operations, 44% lower write cost in LFS, and a 56% increase in the number of concurrent streams serviceable on a video server.

5.1 Experimental setup

Most experiments described in this section were performed on two disks that support zero-latency access (Quantum Atlas 10K and Quantum Atlas 10K II) and two disks that do not implement it (Seagate Cheetah X15 and IBM Ultrastar 18 ES). The disks were attached to a 550 MHz Pentium III-based PC. The

[1] 32 blocks is a representative value for several system variables, and in practice can be smaller on systems with limited resources or larger on systems with custom kernels.

Atlas 10K II was attached via an Adaptec Ultra160 Wide SCSI adapter, the Atlas 10K and Ultrastar were attached via an 80 MB/s Ultra2 Wide SCSI adapter, and the Cheetah via a Qlogic FibreChannel adapter. We also examined workloads with the DiskSim disk simulator [13] configured to model the respective disks. Examining these disks in simulation enables us to quantify the individual components of the overall response time, such as seek and bus transfer time.

5.2 Basic performance

Two workloads, *onereq* and *tworeq*, are used to evaluate basic track-aligned performance. Each workload consists of 5000 locality-free requests within the first zone of the disk. The difference is that *onereq* keeps only one outstanding request at the disk, whereas *tworeq* ensures one request is always queued at the disk in addition to the one being processed.

We compare the efficiency of both workloads by measuring the average per-request *completion time*. We define the completion time as the reciprocal of request throughput (I/Os per second). Therefore, higher disk efficiency will result in a shorter average completion time, all else being equal. We introduce completion time as a metric because it allows us to identify component delays more easily.

For *onereq* requests, completion time equals disk response time, as observed by the device driver, because the next request is not issued until the current one is complete. As usual, disk response time measures the elapsed time from when a request is sent to the disk to when completion is reported. For *onereq* requests, this time includes some fraction when the read/write head is idle because the only outstanding request is waiting for a bus transfer to complete. For *tworeq* requests, the completion time includes only media access delays, since bus activity for any one request is overlapped with positioning and media access for another. The components of completion times for the *onereq* and *tworeq* workloads are shown graphically in Figure 6(a).

5.2.1 Read performance

Figure 5 shows the improvement given by track-aligned accesses on the zero-latency Atlas 10K II. Completion times for track-aligned accesses in *onereq* and *tworeq* decrease by 18% and 32% respectively, which correspond to increases of 22% and 47% in efficiency. The *tworeq* efficiency increase exceeds that of *onereq* because *tworeq* overlaps the bus transfer of the previous request with the media transfer of the current request.

Because of the complete overlap of bus and media transfers, the completion time for a track-aligned track-sized request in the *tworeq* workload is 8.3 ms (calculated as shown in Figure 6(a)). Subtracting 2.2 ms average seek time from the completion time yields 6.1 ms. This observed value corresponds to a single revolution of 6 ms plus 0.1 ms of overhead and confirms that track-aligned accesses to zero-latency disks allow us to read a full track in a single revolution with no rotational latency.

The command queueing of *tworeq* is needed in current systems to address the in-order bus delivery requirement. That is, even though zero-latency disks can read data out of order, they can only send data over the bus in ascending LBN order. This results in only a 3% overlap, on average, between the media transfer and bus transfer for the track-aligned access bar in Figure 6(b). The overlap would be nearly complete if out-of-order bus delivery were used instead, as shown by the bottom bar. Out-of-order bus delivery would improve the efficiency of *onereq* to nearly that of *tworeq* while relaxing the queueing requirement (shown as the "zero bus transfer" curve in Figure 5). Although the SCSI specification allows out-of-order bus delivery using the MODIFY DATA POINTER command, the authors are not aware of any disk vendors that support this operation.

5.2.2 Write performance

The improvement of track-aligned accesses over unaligned ones is larger for writes. For the *onereq* workload on the Atlas 10K II, the completion time of track-sized writes is 10.0 ms for track-aligned access and 13.9 ms for unaligned access, which is a reduction of 28%. For *tworeq*, the reduction in completion time is 34%. These correspond to efficiency increases of 39% and 51% respectively.

This larger improvement, relative to reads, occurs because the seek and bus transfer are completely overlapped. The disk can initiate the seek as soon as the write command reaches it without waiting for the data. While the seek is in progress, the data is transferred to the disk and buffered. Since the average seek

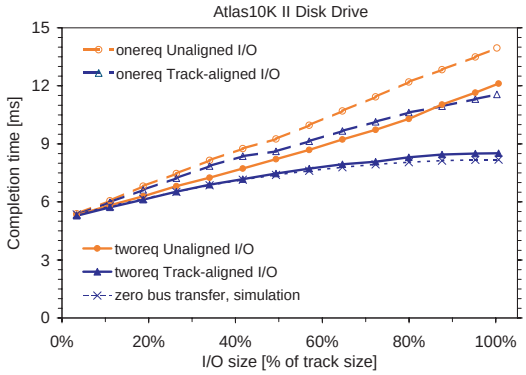
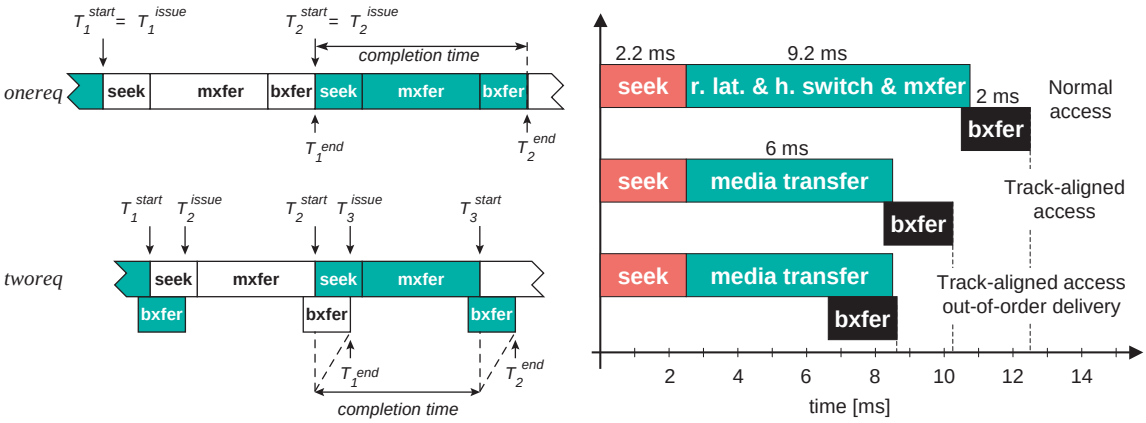


Figure 5: **Average completion time for track-aligned and unaligned reads as a function of I/O size for Quantum Atlas 10K II.** The dashed and solid lines show the measured response time for 5000 random track-aligned and unaligned reads to the disk's first zone for the *onereq* and *tworeq* workloads. There are 528 sectors per track in this zone. The thin dotted line represents the *onereq* workload replayed on a simulator configured with zero bus transfer time; note that it approximates *tworeq* without having to ensure queued requests at the disk.



(a) **Computing completion time.** The completion time of a *tworeq* request is obtained by subtracting T_1^{end} from T_2^{end} which are measured as the end times for requests 1 and 2. For *onereq*, the completion time is $T_2^{end} - T_2^{issue}$. T^{issue} is the time when the request is issued to the disk and T^{start} is the time the request is actually processed at the disk. Notice that for *tworeq*, T^{issue} does not correspond to T^{start} because of queueing at the disk.

(b) **The breakdown of measured response time for a zero-latency disk.** The normal access represents the average response time for track-unaligned access to zero-latency disk and includes seek, rotational latency, and head switch. In the track-aligned access case, the in-order bus transfer (labeled *bxfer*) does not overlap media transfer (*mxfer*). Using out-of-order bus delivery, nearly a complete overlap of bus transfer is possible.

Figure 6: **Elimination of bus transfer time from request response time.**

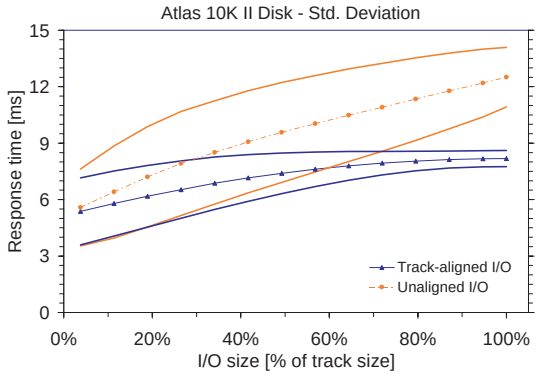


Figure 7: **Response time and its standard deviation for track-aligned and unaligned disk access.** The dotted lines represent the average response time while the envelope of solid lines is the response time ± one standard deviation. The data shown in the graph was obtained by running the *onereq* workload on a simulator configured with zero bus transfer to eliminate the response time variance due to in-order bus delivery.

for the *onereq* workload is 2.2 ms and the data transfer takes about 2 ms, all of the data arrives at the disk before the seek is complete and the zero-latency write begins.

5.2.3 Importance of zero-latency access

The completion time reduction for the other zero-latency disk (the Atlas 10K) is 16% and 32% for track-sized requests in the *onereq* and *tworeq* workloads, corresponding to 19% and 47% higher efficiencies. The difference between this disk and the Atlas 10K II is due to the slightly larger average seek of 2.4 ms.

Completion time does not drop significantly for track-aligned accesses on disks that do not support zero-latency access: The Ultrastar's completion time is reduced by only 6% while the Cheetah shows 8% reduction. These 6–8% reductions in completion times are due solely to the elimination of the head switch penalty in track-aligned access—the rotational latencies of 4 ms (Ultrastar) and 2 ms (Cheetah) are still incurred.

5.2.4 Response time variance

Track-aligned access can significantly lower the standard deviation, σ, of response time as seen in Figure 7. As the request size increases from one sector to the track size, $\sigma_{aligned}$ decreases from 1.8 ms to 0.4 ms, whereas $\sigma_{unaligned}$ decreases from 2.0 ms to 1.5 ms. The standard deviation of the seeks in this workload is 0.4 ms, indicating that the response time variance for aligned access is due entirely to the seeks.

Lowering variance is important for improving the predictability of response time in real time applications. Lowering the overall variance allows applications to make tighter bounds on the worst-case disk access time and therefore schedule work more efficiently.

5.3 FFS experiments

Three simple experiments compare our prototype traxtent-aware FFS performance to that of unmodified FFS. Each test is performed on a freshly-booted system with a clean disk partition.

The first experiment is an I/O-bound linear scan through a 4 GB file. As expected, the traxtent-aware system runs 5% slower than unmodified FFS (199.8 s vs. 189.6 s). This is because FFS is optimized for large sequential single-file access and reads at the maximum disk streaming rate, whereas the traxtent system inserts an excluded block one out of every twenty blocks (5%).

The second experiment consists of the `diff` application comparing two large files. Because `diff` interleaves fetches from the two files, we expect to see a large speedup from improved disk efficiency. For 512 MB files, the traxtent-aware system completes 19% faster than unmodified FFS. A more detailed analysis shows that traxtent FFS performs 6724 I/Os (average size of 160 KB) in 56.6 s while the unmodified FFS performs

only 4108 I/Os (average size of 256 KB) but requires 69.7 s. To compare these numbers, we normalize the accesses by subtracting out the media transfer time and bus transfer time and find that the unmodified FFS incurs 6.9 ms of overhead per request (including seek, rotational latency, and track switch time) while the traxtent implementation incurs only 2.2 ms of overhead per request. The 19% improvement in overall completion time corresponds to an improvement in disk efficiency of 23%, which matches exactly with the predicted value from the Atlas 10K model for this request pattern.

The third experiment verifies write performance by copying a 1 GB file to another file in the same directory. FFS commits dirty buffers as soon as a complete cluster is created, which results in two interleaved request streams to the disk. This test shows a 20% reduction in run time for the traxtent-aware system (124.9 s vs. 156.9 s).

5.4 Log-structured File System

The log-structured file system (LFS) [27] was designed to reduce the cost of disk writes. Towards this end, it remaps all new versions of data into large, contiguous regions called segments. Each segment is written to disk with a single I/O operation, amortizing the positioning cost over one large write. A significant challenge for LFS is ensuring that empty segments are always available for new data. LFS answers this challenge with an internal defragmentation operation called *cleaning*. Cleaning of a previously written segment involves identifying the subset of "live" blocks, reading them into memory, and writing them into a new segment. Live blocks are those that have not been overwritten or deleted by later operations.

There is a performance trade-off between write efficiency and the cost of cleaning. Larger segments offer higher write efficiency but incur larger cleaning cost since more data has to be transferred for cleaning [18, 31]. Additionally, the transfer of large segments hurts the performance of small synchronous reads [5, 18]. Therefore, the goal is to find a segment size that is optimal for write efficiency, cleaning cost, and small synchronous I/O performance.

To evaluate the benefit of using track-based access for LFS segments, we use the overall write cost (OWC) metric described by Matthews et al. [18], which is a refinement of the *write cost* metric defined for the Sprite implementation of LFS [27]. It expresses the cost of writes in the file system assuming all data reads are serviced from the system cache. The OWC metric is defined as the product of write cost and disk transfer inefficiency:

$$\begin{aligned} OWC &= \mathit{WriteCost} \times \mathit{TransferInefficiency} \\ &= \frac{N_{written}^{new} + N_{read}^{clean} + N_{written}^{clean}}{N_{written}^{data}} \times \frac{T_{xfer}^{actual}}{T_{xfer}^{ideal}} \end{aligned}$$

where N is the number of segments written due to new data or read and written due to segment cleaning and T is the time for one segment transfer. *WriteCost* depends on the workload (i.e., how much new data is written and how much old data is cleaned) but is independent of disk characteristics. *TransferInefficiencyI*, on the other hand, depends only on disk characteristics. Therefore, we can use the values for *WriteCost* given by Matthews et al. for their Auspex server trace [18] and determine the OWC for current disks based on measured transfer inefficiency. Based on data in Figure 1, the *TransferInefficiency* is the ratio between the maximum utilization and the achieved utilization.

Figure 8 shows that the OWC is smaller when track-aligned disk access is used and the cost is minimized when the segment size matches the track size. Unlike our use of empirical data for determining *TransferInefficiency*, Matthews et al. estimate its value as

$$\mathit{TransferInefficiency} = T_{pos} \times \frac{BW_{disk}}{S_{segment}} + 1$$

where $S_{segment}$ is the size of a segment (in bytes) and T_{pos} is the average positioning time (i.e., seek and rotational latency). To show that our results are in agreement with their findings, we computed OWC for the Atlas 10K II based on its specifications and plotted it in Figure 8 (labeled "5.2 ms*41 MB/s") along with the OWC values for the track-aligned and unaligned I/O. Because the empirical values were obtained from the disk's first zone, we used the average seek of 2.2 ms for the first zone, rotational latency of 3 ms and peak bandwidth of 41 MB/s instead of the average values of 4.7 ms seek and 32 MB/s bandwidth. As expected, the model is a good match for the unaligned case.

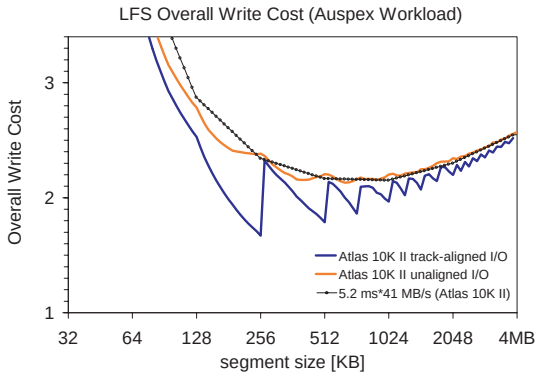


Figure 8: **LFS overall write cost for the Auspex trace as a function of segment size.** The two solid lines represent track-aligned and unaligned disk access to Atlas 10K II's first zone. The dotted line is the overall write cost for the Atlas 10K II disk computed with the transfer inefficiency model as described in Matthews et al. [18].

5.4.1 Variable segment size

As shown in Figure 8, the lowest write cost is achieved when the size of a segment matches the size of a track. Unfortunately, as discussed in Section 3, tracks have different numbers of sectors depending on their location on the disk and any defect management changes. An LFS must therefore allow for variable segment sizes in order to match segment boundaries to track boundaries.

A segment usage table records whether each segment is allocated or free. In the SpriteLFS implementation [27], this table is kept as an in-memory kernel structure and is stored in the checkpoint region of the file system. The BSD-LFS implementation [30] stores this table in a special file called the IFILE. Because of the frequent use of this file, the segment usage table is almost always in the file system's buffer cache.

Supporting variable-sized segments is relatively straightforward. This can be done by augmenting the per-segment information in the segment usage table with the segment starting location (the LBN) and the segment length. If segment starting locations must be block-aligned and/or segment lengths must be multiples of the block size, the segment can be pruned as is done in the traxtent-aware FFS implementation described in Section 4.

Using variable-sized segments also requires minimal changes. During the creation of the file system, each segment's starting location and length is set according to the track boundary information. When a new segment is allocated in memory, its size is looked up in the segment usage table. When the segment becomes full, it is written to the disk at the starting location given in the segment usage table. The procedures for reading segments and for cleaning are similar.

5.5 Video servers

A video server is designed to serve large numbers of video streams to clients at guaranteed rates. To accomplish this, the server fetches a time interval of video (i.e., 0.5 s) for all streams. This time interval is called a *round*. Then, while those streams are transferred to the clients from the server's buffer, the server schedules the next round of requests. Since the per-stream disk access time is much less than the round time, many simultaneous streams can be supported by a single disk.

The round time in a video server is determined by several factors: the video bit rate (which determines the individual I/O size), the number of streams that can be supported by all disks (determined by disk access time and bandwidth), and the amount of available buffer space. The buffer space required at the server is $2 \times IOsize_{\text{disk}} \times V$, where V is the number of simultaneous video streams. Round time R also determines the startup latency of a newly admitted stream. Assuming the video server stripes data across D disks, the worst-case startup latency is $R \times (D + 1)$ [24, 8, 28, 32]. In practice, round time is chosen to meet system goals given a trade-off between startup latency and the maximum number of supportable streams.

5.5.1 Model

We evaluate the benefits of track-aligned access in video servers with a model that, for a given round time, determines the maximum number of streams supported by the video server. Our evaluation is done for a video server with 10 Quantum Atlas 10K II disks and follows the evaluation of the disk striping model presented for the RIO video server [28].

Given a specific round time, the number of streams per disk N_{disk} is computed as

$$N_{disk} = \frac{R}{T_{stream}}$$

where T_{stream} is the service time per stream. This per-stream service time will limit the maximum number of streams that a single disk can serve and is computed as

$$T_{stream} = \frac{IOsize_{disk}}{BW_{disk} \times Efficiency_{disk}}$$

where $IOsize_{disk}$ is the product of round time and video bit rate and $Efficiency_{disk}$ is a function of I/O sizes as illustrated in Figure 1. Since track-aligned access increases $Efficiency_{disk}$, more streams can be serviced by the disk.

5.5.2 Soft real-time

Several video server projects, such as Tiger [3] and RIO [28], have designed video servers using soft real-time guarantees. These systems provide guarantees that, with a certain probability, a request will not miss its deadline. This allows a relaxation on the assumed worst-case seek and rotational latency and results in higher bandwidth utilization for both track-aligned and unaligned access.

To evaluate our model system, we measured the performance of an Atlas 10K II disk. We issued a fixed number of random track-sized requests simultaneously and measured how long they took to complete. This measurement was repeated 10,000 times for each number of simultaneous requests from 10 to 80. 80 is the maximum number of simultaneous 4 Mb/s streams that can be supported by a disk, given a 41 MB/s peak bandwidth of the disk.

From the PDF of the measured response times, we obtained the round time that would meet 99.99% of the deadlines for the 4 Mb/s rate. Given a 0.5 s round time (which translates to a worst-case startup latency of 5.5 s for the 10-disk striped array), the track-aligned system is able to support up to 70 streams per disk. In contrast, the unaligned system is only able to support 45 streams per disk. Therefore, the track-aligned access can support 56% more streams at this minimal startup latency.

To support more than 70 and 45 streams per disk for the track-aligned and unaligned systems, the I/O size must increase to meet the real-time requirements of the video streams. This increase in I/O size, however causes an increase in the round time, which in turn increases the startup latency as shown in Figure 9. At 70 streams per disk, the startup latency for the track-aligned system is 4× smaller than for the track-unaligned system.

5.5.3 Hard real-time

Although many video servers implement soft real-time requirements, there are applications that require hard real-time deadlines. In their admission control algorithms, systems must assume the worst-case response time to ensure that no deadline is missed. In computing the worst-case response time, one assumes the worst-case seek, transfer time, and rotational latency. Both the track-aligned and unaligned access systems have the same values for the worst-case seek[2]. However, the worst-case rotational latency for track-unaligned access is one revolution, whereas track-based access suffers no rotational latency. The worst-case transfer time will be similar except that the track-unaligned system must assume a head switch will occur. Using the model

[2]The worst-case seek time for a single stream is much smaller than a full strobe seek (seek from one edge of the disk to the other) and improves with increasing number of streams serviced by the disk. This is because a C-LOOK scheduler (either inside or outside the disk) will order all requests to the disk in one round to minimize the seek distance. Therefore, the worst-case seek time charged to a stream is equal to the longest possible seek route taken to serve all streams in one full sweep of the head divided by the number of streams.

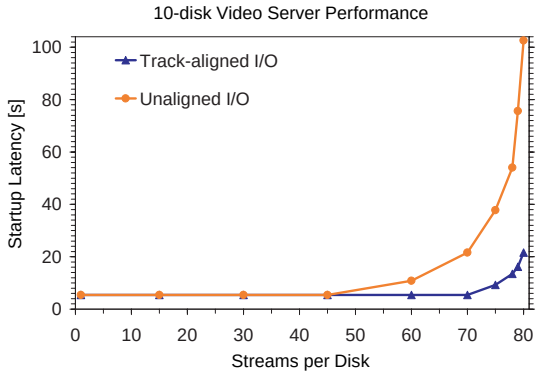


Figure 9: **Worst-case startup latency of a video stream for track-aligned and unaligned accesses as a function of streams per disk.** The startup latency is shown for a 10-disk array. Thus, given that the maximum number of simultaneous streams supported by a single disk is 80, the 10-disk array can support up to 800 streams. The 10-disk array evaluation of startup latency follows the evaluation of RIO video server [28].

from Section 5.5.1 with a 4 Mb/s bit rate and an I/O size of 264 KB, the track-unaligned system supports 36 streams per disk whereas the track-based system supports up to 67 streams. This translates into 45% and 83% disk efficiency, respectively. With an I/O size of 528 KB, unaligned access yields 52 streams vs. 75 for track-based system. Unaligned I/O size must exceed 2.5 MB and incur a startup latency of 60.5 s to achieve the same efficiency as the track-aligned system.

6 Additional related work

Much related work has been discussed throughout this paper. Some other notable related work has promoted zone-based allocation and detailed disk-specific request generation for small requests.

The Tiger video server [3] allocated primary copies of videos to the early portions of disks' LBN space in order to exploit the higher bandwidth of outer zones. Secondary copies were allocated to the lower bandwidth zones. Van Meter [21] suggested that there was general benefit in changing file systems to understand that different regions of the disk provide different bandwidths.

By utilizing even more detailed disk information, several researchers have shown substantial decreases in small request response times [9, 39, 41]. For small writes, these systems detect the position of the head and re-map small writes to the nearest free block to minimize the positioning costs [9, 39]. For small reads, the SR-Array [41] determines the head position when the read request is to be serviced and reads the closest of several replicas.

7 Summary

This paper presents a case for track-aligned extents. It demonstrates feasibility with a working prototype, and it demonstrates value with direct measurements. At the low level, traxtent accesses are shown to increase disk efficiency by approximately 50% compared to track-unaligned accesses of the same size. At the system level, traxtents are shown to increase application efficiency by 25–56% for large file workloads, video servers, and write-bound log-structured file systems.

References

[1] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *ACM Symposium on Operating System Principles* (Asilomar, Pacific Grove, CA). Published as *Operating Systems Review*, **25**(5):198–212, 13–16 October 1991.

[2] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. *Annual USENIX Technical Conference* (New Orleans, LA, 16–20 January 1995), pages 277–288. USENIX Association, 1995.

[3] William J. Bolosky, Joseph S. Barrera, Richard P. Draves, Robert P. Fitzgerald, Garth A. Gibson, Michael B. Jones, Steven P. Levi, Nathan P. Myhrvold, and Richard F. Rashid. *The Tiger video fileserver.* Technical Report MSR–TR–96–09. Microsoft Corporation, April 1996.

[4] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel.* O'Reilly & Associates, 2001.

[5] Scott Carson and Sanjeev Setia. Optimal write batch size in log-structured file systems. *USENIX Workshop on File Systems* (Ann Arbor, MI), pages 79–91, 21–22 May 1992.

[6] Edward Chang and Hector Garcia-Molina. Reducing initial latency in a multimedia storage system. *International Workshop on Multi-Media Database Management Systems* (Blue Mountain Lake, NY), pages 2–11, 14–16 August 1996.

[7] Edward Chang and Hector Garcia-Molina. Effective memory use in a media server. *VLDB* (Athens, Greece.), pages 496–505, 26–29 August 1997.

[8] Ann L. Chervenak, David A. Patterson, and Randy H. Katz. Choosing the best storage system for video service. *ACM Multimedia 95* (San Francisco, CA USA), pages 109–119, 5–9 November 1995.

[9] Tzi-Cker Chiueh and Lan Huang. *Trail: track-based logging in Stony Brook Linux.* Technical report ECSL TR-68. SUNY Stony Brook, December 1999.

[10] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (Alanta, Georgia), pages 59–70, 1–4 May 1999.

[11] Eran Gabber and Elizabeth Shriver. Lets put NetApp and CacheFlow out of business. *SIGOPS European Workshop* (Kolding, Denmark), pages 85–90, 17–20 Sept. 2000.

[12] 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.

[13] Gregory R. Ganger, Bruce L. Worthington, and Yale N. Patt. *The DiskSim Simulation Environment Version 1.0 Reference Manual*, CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.

[14] Shahram Ghandeharizadeh, Seon Ho Kim, and Cyrus Shahabi. *On configuring a single disk continuous media server.* Technical report 94–590. USC., 8 November 1994.

[15] Sanjay Ghemawat. *The modified object buffer: a storage management technique for object-oriented databases.* PhD thesis. Massachusetts Institute of Technology, Cambridge, MA, 7 September 1995.

[16] Dominic Giampaolo. *Practical file system design with the Be file system.* Morgan Kaufmann, 1998.

[17] Kimberly Keeton and Randy H. Katz. The evaluations of video layout strategies on a high-bandwidth file server. *4th International Workshop on Network and Operating System Support for Digital Audio and Video* (Lancaster, England, UK.), pages 228–229, 3–5 November 1993.

[18] 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.

[19] 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.

[20] 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.

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

[22] Rajeev Nagar. *Windows NT File System Internals: A Developer's Guide*. O'Reilly & Associates, 1997.

[23] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. *ACM Symposium on Operating System Principles* (Orcas Island, WA). Published as *Operating Systems Review*, **19**(5):15–24, December 1985.

[24] Banu Ozden, Rajeev Rastogi, and Avi Silberschatz. Disk striping in video server environments. *International Conference on Multimedia Computing and Systems* (Hiroshima, Japan), pages 580–589, 17–23 June 1996.

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

[26] ReiserFS. http://devlinux.com/projects/reiserfs/.

[27] 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.

[28] Jose Renato Santos, Richard R. Muntz, and Berthier Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. *ACM SIGMETRICS 2000* (Santa Clara, CA). Published as *Performance Evaluation Review*, **28**(1):44–55, 17–21 June 2000.

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

[30] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An implementation of a log-structured file system for UNIX. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 307–326, January 1993.

[31] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File system logging versus clustering: a performance comparison. *Annual USENIX Technical Conference* (New Orleans), pages 249–264. Usenix Association, 16–20 January 1995.

[32] Prashant J. Shenoy and Harrick M. Vin. Efficient striping techniques for multimedia file servers. *7th International Workshop on Network and Operating System Support for Digital Audio and Video* (St. Louis, MO), pages 25–36, 19–21 May 1997.

[33] Tracy F. Sienknecht, Rich J. Friedrich, Joe J. Martinka, and Peter M. Friedenbach. The implications of distributed data in a commercial environment on the design of hierarchical storage management. *Performance Evaluation*, **20**(1–3):3–25, May 1994.

[34] Keith Smith and Margo Seltzer. *File layout and file system performance*. Technical report TR-35-94. Harvard University, December 1994.

[35] Keith A. Smith and Margo Seltzer. A comparison of FFS disk allocation policies. *USENIX.96* (San Diego, CA., 22–26 January 1996), pages 15–25. USENIX Assoc., 1996.

[36] Adam Sweeney. Scalability in the XFS file system. *USENIX.* (San Diego, California), pages 1–14, 22–26 January 1996.

[37] Nisha Talagala, Remzi H. Dusseau, and David Patterson. *Microbenchmark-based extraction of local and global disk characteristics*. Technical report CSD–99–1063. University of California at Berkeley, 13 June 2000.

[38] Werner Vogels. File system usage in Windows NT 4.0. *ACM Symposium on Operating System Principles* (Kiawah Island Resort, Charleston, South Carolina, 12–15 December 1999). Published as *Operating System Review*, **33**(5):93–109. ACM, December 1999.

[39] Randolph Y. Wang, David A. Patterson, and Thomas E. Anderson. Virtual log based file systems for a programmable disk. *Symposium on Operating Systems Design and Implementation* (New Orleans, LA, 22–25 February 1999), pages 29–43. ACM, February 1999.

[40] 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.

[41] 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.