Conference on File and Storage Technologies (FAST) January 28-30, 2002. Monterey, CA.

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

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

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 some uses of traxtents in file systems and video servers. For large-file workloads, a version of FreeBSD's FFS implementation that exploits traxtents reduces application run times by up to 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.

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 achieving 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 by issuing fewer, larger disk requests. This is usually done with only a vague understanding of disk characteristics, focusing on the notion that bigger requests are better because they amortize per-request positioning delays over larger data transfers. Although this notion is generally correct, there are performance and complexity costs associated with making requests larger and larger. For video servers, ever-larger

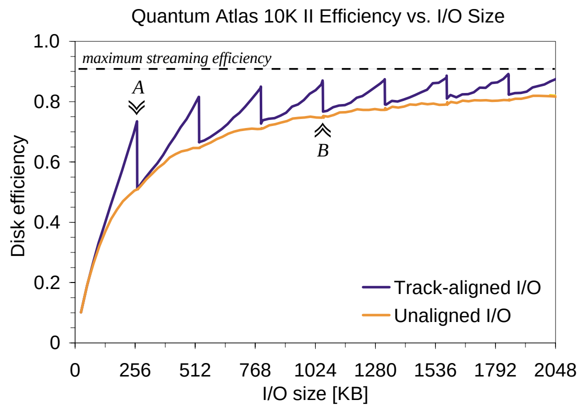


Figure 1: **Measured advantage of track-aligned access over unaligned access.** *Disk efficiency* is the fraction of total access time spent moving data to or from the media. The maximum streaming efficiency is less than 1.0, because no data is transferred when switching from one track to the next. The track-aligned and unaligned lines show disk efficiency for random, constant-sized reads within a Quantum Atlas 10K II's first zone (264 KB per track). Point A highlights the higher efficiency of track-aligned access (0.73, or 82% of the maximum) over unaligned access for a track-sized request. Point B shows where unaligned I/O efficiency catches up to the track-aligned efficiency at Point A. The peaks in the track-aligned curve correspond to multiples of the track size.

requests increase both buffer space requirements and stream initiation latency [6, 7, 22]. Log-structured file systems (LFS) incur higher cleaning overheads as segment size increases [5, 24, 33]. Even for general file system operation, allocation of very large sequential regions competes with space management robustness [25], and very large accesses may put deep prefetching ahead of foreground requests. Also, large requests can be used for small files by grouping their contents [14, 15, 17, 32, 33], but larger requests require grouping more files with weaker inter-relationships. 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, quantified as the the fraction of total access time spent moving data to or from the media.

The efficiency improvement stems from two main sources. First, track-aligned access minimizes the number of track switches, whose times have not decreased 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.

The key challenge with exploiting disk-specific knowledge is clean, robust integration: complexity must be minimized, systems must not become tied to specific devices, and system management must not be made harder. These concerns can be addressed by minimizing the disk-specific details needed, determining them automatically for any disk, and incorporating them in a generic fashion. This paper promotes the use of track boundaries, describes algorithms for detecting them automatically, and describes how they can be cleanly integrated into existing systems. In particular, simply changing a file system to support variable-sized extents is sufficient — the file system code need not depend on any particular disk's track boundaries. Further, variable-sized extents allow a file system to accomodate other boundary-related goals, such as matching writes to stripe boundaries in order to avoid RAID 5 read-modify-write operations [9].

This paper extensively explores track-based access. Detailed disk measurements show increased disk efficiency and reduced access time variance. They also identify system requirements that must be satisfied to achieve the highest efficiency. A prototype implementation of a traxtent-aware FFS file system in FreeBSD 4.0 illustrates the minimal changes needed and the resulting benefits. For example, when accessing two large files concurrently, the traxtent-aware FFS yields 20% higher performance compared to current defaults. For streaming media workloads, a video server can support either 56% more concurrent streams at the same startup latency or a 5× reduction in startup latency and buffer space at the maximum number of concurrent streams. Finally, we compute 44% lower overall write cost for track-sized segments in LFS.

Although track boundary knowledge was used for allocation and access decisions in some pre-SCSI systems, no current or recent system that we are aware of does so. This paper makes several enabling contributions:

1. It identifies and quantifies the benefits of track-based access on modern disks, showing up to 50% increases in efficiency. This is a compelling performance boost.

2. It introduces new algorithms for automatically detecting track boundaries. This task is more difficult than might be expected, because of zoned recording and media defect management.

3. It describes a minimal set of changes needed to use track boundary knowledge in an existing file system. This experience supports our contention that exploiting disk-specific knowledge appropriately need not introduce hardware dependences.

4. It shows that the disk efficiency benefits translate into application performance increases in some real cases, including streaming media services and large file manipulations.

The remainder of this paper is organized as follows. Section 2 motivates track-based access by describing the technology trends 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.

2 Track-based Disk Access

In determining what data to read and write 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, resource limitations and imperfect information about future accesses impose costs on the use of very large requests.

This section discusses the system-level issues 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

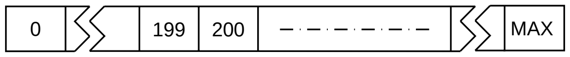
Four system-level factors oppose the use of ever-larger requests: (1) responsiveness, (2) limited buffer space, (3) irregular access patterns, and (4) storage space management.

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 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) can result in substantial I/O wait times for small, synchronous requests. This problem has been noted for both FFS and LFS [5, 37]. 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 fetch video segments in carefully-scheduled 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 5.4 quantifies the start-up latency required for modern disks.

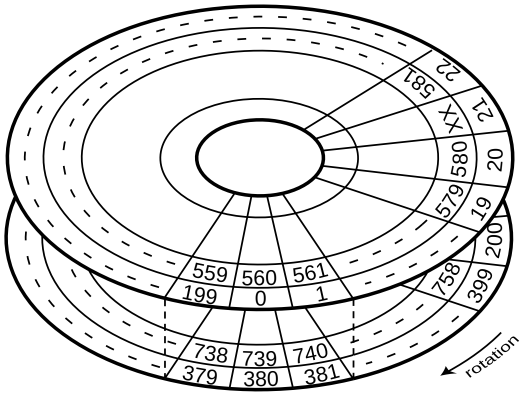
Buffer space. Although memory sizes continue to grow, they remain finite. Larger disk requests stress memory resources in two ways. For reads, larger disk requests are usually created by fetching more data farther in advance of the actual need for it; this prefetched data must be buffered until it is needed. For writes, larger disk requests are usually created by holding more data in a 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, 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 [1, 29, 45], 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 [15, 40] and below 64 KB in Microsoft Windows systems [12, 45]. 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 [14, 15, 17, 32, 33], 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. As the target request size grows, 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, growth of individual data objects must be accommodated. All of these needs must be met with little or no information about future storage allocation and deallocation operations. Collectively,



(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 to maximize streaming bandwidth. The picture also shows a defect between the sectors with LBNs 580 and 581, depicted as XX, 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	512B Sectors per Track	Number of Tracks	Capacity
HP C2247	1992	5400	1 ms	10 ms	96–56	25649	1 GB
Quantum Viking	1997	7200	1 ms	8.0 ms	216–126	49152	4.5 GB
IBM Ultrastar 18 ES	1998	7200	1.1 ms	7.6 ms	390–247	57090	9 GB
IBM Ultrastar 18LZX	1999	10000	0.8 ms	5.9 ms	382–195	116340	18 GB
Quantum Atlas 10K	1999	10000	0.8 ms	5.0 ms	334–224	60126	9 GB
Seagate Cheetah X15	2000	15000	0.8 ms	3.9 ms	386–286	103750	18 GB
Quantum Atlas 10K II	2000	10000	0.6 ms	4.7 ms	528–353	52014	9 GB

Table 1: **Representative disk characteristics.** Note the small change in head switch time relative to other characteristics.

these facts create a complex storage management problem. Systems can address this problem with combinations of pre-allocation heuristics [4, 18], on-line reallocation actions [23, 33, 41], and idle-time reorganization [2, 24]. There is no straightforward solution and the difficulty grows with the target disk request size, because more related data must be clustered.

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.

Head switch. A head switch occurs when a single request accesses a sequence of LBNs whose on-disk locations span two tracks. This head switch consists of turning on the electronics for the appropriate read/write head and adjusting its position to account for inter-surface alignment imperfections. The latter step requires the disk to read servo information to determine the head's location and then to shift the head towards the center of the second track. In the example of Figure 2(b), head switches occur between LBNs 199 and 200, 399 and 400, and 598 and 599.

Even compared to other disk characteristics, head switch time has improved little in the past decade. While disk rotation speeds have improved by 3$\times$ and average seek times by 2.5$\times$, head switch times have decreased by only 20–40% (see Table 1). At 0.6–1.1 ms, a head switch now takes about 1/5 of a revolution for a 15,000 RPM disk. This trend has increased the significance of head switches. Further, this trend is expected to continue, because rapid decreases in inter-track spacing require increasingly precise head positioning.

Naturally, not all requests span track boundaries. The probability of a head switch, P_{hs}, depends on workload and disk characteristics. For a request of N sectors and a track size of SPT sectors, $P_{hs} = (N-1)/SPT$, assuming that the requested locations are uncorrelated with track boundaries. For example, with 64 KB requests ($N = 128$) and an average track size of 192 KB ($SPT = 384$), a head switch occurs for every third access, on average. With N approaching SPT, 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, track-aligned access improves the response time of most requests by the 0.6–1.1 ms head switch time.

Zero-latency access. A second disk feature that pushes for track-based access is zero-latency access, also known as immediate access or access-on-arrival. When disk firmware wants to read N 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 N sectors in ascending LBN order. With zero-latency access support, disk firmware can read the N 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 involved because all sectors on the track are needed. The N 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 moved from host memory to the disk's buffers before it can be written onto the media.

As an example of zero-latency access on the disk from Figure 2(b), consider a read request for LBNs 200–399.

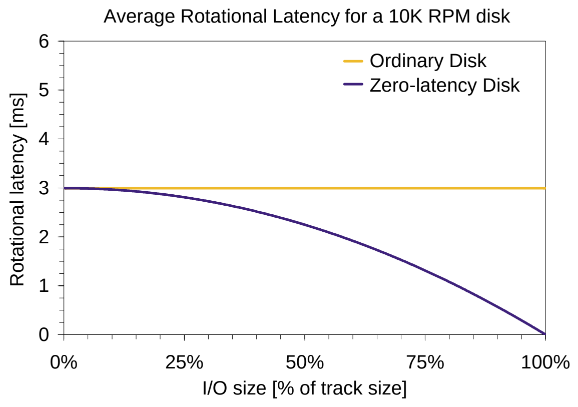


Figure 3: **Average rotational latency for ordinary and zero-latency disks as a function of track-aligned request size.** The request size is expressed as a percentage of the track size.

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 immediately read LBNs 380–399. It then reads the sectors with LBNs 200–379. In this way, the entire track can be 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 increases, as shown in Figure 3. Therefore, a request to the zero-latency access disk for all SPT sectors on a track requires only one revolution after the seek. An ordinary disk, on the other hand, has an expected rotational latency of $(SPT-1)/(2 \cdot SPT)$, or approximately 1/2 revolution, regardless of the request size and thus a request requires anywhere from one to two (average of 1.5) revolutions.

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 SPT sectors involves four delays: seek, rotational latency, SPT sectors worth of media transfer, and head switch. An SPT-sector track-aligned access 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, a system that sequentially reads a single large file will also see little benefit, because positioning costs can be amortized over megabyte sized transfers and the disk's prefetching logic will ensure that this occurs. Track-based access provides the highest benefit to applications with medium-sized I/Os. One set of examples is streaming media services, such as video servers, MP3 servers, and CDN caches. Another includes storage components (e.g., Network Appliance's filers [19], HP's AutoRAID [47], or EMC's Symmetrix) that map data to disk locations in mid-sized chunks. Section 5 explores several concrete examples of such applications.

3 Traxtent-aware System Design

Track-based disk access is a design option for any system component that allocates disk locations and generates disk requests. In some systems, like the one used in our experiments, these decisions are made in the system software (e.g., file system) of a workstation, file server, or content-caching appliance. In others, the system software decisions are overridden by a logical disk [11] or a high-end disk array controller [42, 47], using some sort of mapping table to translate requested LBNs to internal disk locations. Track-based disk access is appropriate within any of these systems, and it requires 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, a system must first obtain it. Specifically, a system must know the range of LBNs that map onto each track. Under ideal circumstances, the disk would provide this information directly. However, since current 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), each with a different number of sectors per track. 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 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 modified to avoid 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 in a spare sector. Slipping is more efficient and more common, but it affects the mappings of subsequent LBNs.

Although track detection can be difficult, it 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 [30].

3.2 Allocation and access

To utilize track boundary information, the algorithms for on-disk placement and request generation must support variable-sized extents. Extent-based file systems, such as NTFS [28] and XFS [43], allocate disk space to files by specifying ranges of LBNs (extents) associated with each file. Such systems lend themselves naturally to track-based alignment of data: during allocation, extent ranges can be chosen to fit track boundaries. Block-based file systems, such as Ext2 [4] and FFS [25], group LBNs into fixed-size allocation units (blocks), typically 4 KB or 8 KB in size.

Block-based systems can approximate track-sized extents by placing sequential runs of blocks such that they never span track boundaries. This approach 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. Alternately, this space can be reclaimed if the cache manager can be modified to handle partially-valid and partially-dirty blocks.

Like any clustering storage system, a traxtent-based system must address aging and fragmentation and the standard techniques apply: pre-allocation [4, 18], on-line reallocation [23, 33, 41], and off-line reorganization [2, 24]. For example, when a 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. The same holds when grouping small files [15, 32]. When the file system becomes aged and fragmented, on-line or off-line reorganization can be used to re-optimize the on-disk layout. Such reorganization can also be used for retrofitting pre-existing disk partitions or adapting to a replacement disk. The point of this paper is that traxtents are a good target layout for these techniques.

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 uncovered an additional design consideration: current systems only realize the full benefit of track-based requests when using command queueing at the disk. Although zero-latency disks can access LBNs on the media in any order, current SCSI and IDE/ATA controllers only allow for in-order delivery to or 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 operating systems 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

The general 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 track boundary is crossed. Starting with sector 0 of the disk ($S = 0$), the 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.). The extractor avoids rotational latency variance by synchronizing with the rotation speed, issuing each request at (nearly) the same offset in the rotational period; rotational latency could also be addressed by averaging many observations, but at a substantial cost in extraction time. Eventually, an N-sector read returns in more time than a linear model suggests (i.e., $N = SPT + 1$), which identifies sector $S+N$ as the start of a new track. The algorithm then repeats with $S = S+N-1$.

The method described above is clearly suboptimal; our actual implementation uses a binary search algorithm to find when $N = SPT + 1$. In addition, once SPT is determined for a track, the common case of each subsequent track being the same size is quickly verified. This verification checks for a discontinuity between $S + SPT - 1$ and $S + SPT$. If so, it sets $S = S + SPT - 1$ and moves on. Otherwise, it sets $N = 1$ and uses the base method; this occurs mainly on the first track of each zone and on tracks containing defects. With these enhancements, the algorithm extracts the track boundaries of a 9 GB disk (a Quantum Atlas 10K) in four hours. Talagala et al. [44] describe a much quicker algorithm that extracts approximate geometry information using just the read command; however, for our purposes, the exact track boundaries must be identified.

One difficulty with using read requests to detect track boundaries is the caching performed by disk firmware. To obviate the effects of firmware caching, the algorithm interleaves 100 parallel extraction operations to widespread disk locations, such that the cache is flushed each time we return to block S. An alternative approach would be to use write requests; however, this is undesirable because of the destructive nature of writes and because some disks employ write-back caching.

4.1.2 SCSI-specific approach

The SCSI command set supports query operations that can simplify track boundary detection. Worthington et al. [48] describe how these operations can be used to determine LBN-to-physical mappings. Building upon their basic mechanisms, we have implemented an automated disk drive characterization tool called DIXtrac [35]. This tool includes a five-step algorithm that exploits the regularity of disk 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), largely 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.

DIXtrac has been successfully used on dozens of disks, spanning 11 different disk models from 4 different manufacturers. Still, it does not always work. In our experience, step #3 has failed several times when we tried a new disk with a previously unknown (to us) mapping scheme — most are now part of DIXtrac's expertise, but future advances may again baffle it. When this happens, a system can fall back on the general approach or, better yet, a SCSI-specific version of it. That is, the general algorithm can be specialized to use SCSI's SEND/RECEIVE DIAGNOSTIC command instead of request timings. Such expertise-free, SCSI-specific extraction of track boundaries requires approximately 2.0–2.3 translations per track for most disks; it requires approximately 5 minutes for the 9GB Atlas 10K.

4.2 Traxtent support in FreeBSD

This section reviews the basic operation of FreeBSD FFS [25] 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 disk addresses 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 device driver during an access. (Device drivers adjust sector numbers to partition boundaries.) In our experiments, the file system block size is 8 KB (sixteen contiguous LBNs). In this section, "block" refers to a physical block.

FFS partitions the set of physical blocks 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

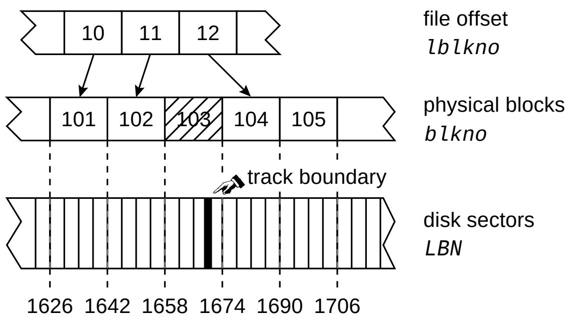


Figure 4: **Mapping system-level blocks to disk sectors.** Physical block 101 maps directly to disk sectors 1626–1641. Block 103 is an *excluded* block (see Section 4.2.2) because it spans the disk track boundary between LBNs 1669–1670.

underlying disk's physical geometry. Although this geometry dependence is no longer real, 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.

FreeBSD's FFS implementation uses the clustered allocation and access algorithms described by McVoy & Kleiman [26]. 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. Usually, the cluster selected consists of the *N* blocks immediately following the last block committed. To assist in fair local allocation among multiple files, FFS 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 read-ahead (a.k.a. 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 read-ahead blocks drops below 32, FFS issues a new read-ahead of length *l* beginning with the first noncached block, 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

This section describes the few, small changes required to integrate traxtent-awareness into FreeBSD FFS.

[1]32 blocks is a representative default value. It may be smaller on systems with limited resources or larger on systems with custom kernels.

Excluded blocks and traxtent allocation. We introduce the concept of the *excluded* block, highlighted in Figure 4. Blocks that span track boundaries are excluded from allocation decisions by marking them as used in the free-block map. 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. As per-track capacity grows, the frequency of excluded blocks decreases—for the Atlas 10K II, one in thirty is excluded.

Traxtent-sized access. No fundamental changes are necessary in the FFS clustered read-ahead algorithm. FFS properly identifies runs of blocks between excluded blocks as clusters and accesses them with a single disk request. Until non-sequential access is detected, we ignore the "sequential count" to prevent multiple partial accesses to a single traxtent; for non-sequential file sessions, the default mechanism is used. We handle the special case where there is no excluded block between traxtents by ensuring that no read-ahead 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 identified, adjusted to the file system's partition, and 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.

5 Evaluating Traxtents

This section examines the performance benefits of track-based access at two levels. First, it evaluates the disk in isolation, finding a 50% improvement in disk efficiency and a reduction in response time variance. Second, it quantifies system-level performance gains, showing a 20% reduction in run time for large file operations, a 56% increase in the number of concurrent streams serviceable on a video server, and a 44% lower write cost for a log-structured file system.

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 (Seagate Cheetah X15 and IBM Ultrastar 18 ES). The disks were attached to a 550 MHz Pentium III-based PC. The Atlas 10K II was attached via

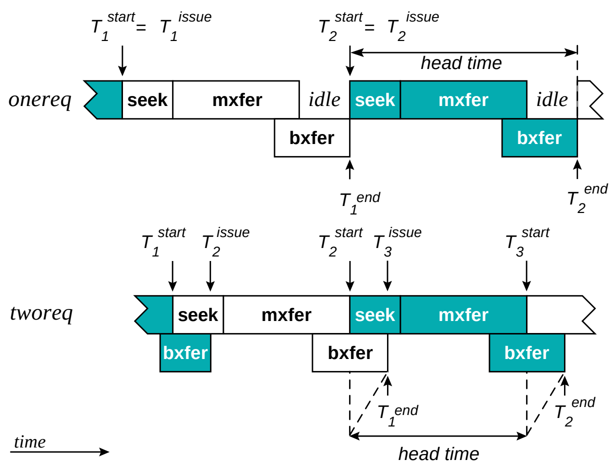


Figure 5: **Expressing head time.** The head time of a *onereq* request is $T_2^{end} - T_2^{issue}$. For *tworeq*, the head time is $T_2^{end} - T_1^{end}$. T^{issue} is the time when the request is issued to the disk, T^{start} is when the disk starts servicing the request, and T^{end} is when completion is reported. Notice that for *tworeq*, T^{issue} does not equal T^{start} because of queueing at the disk.

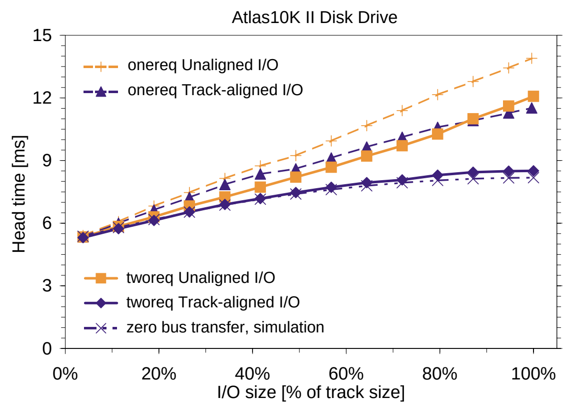


Figure 6: **Average head time for track-aligned and unaligned reads for Quantum Atlas 10K II.** The dashed and solid lines show the average of measured times for 5000 random track-aligned and unaligned reads to the disk's first zone for the *onereq* and *tworeq* workloads. Multiple runs for a subset of the points reveal little variation (<0.4%) between average head times for distinct sets of 5000 random requests. 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.

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 [16] 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 Disk performance

Two workloads, *onereq* and *tworeq*, are used to evaluate basic track-aligned performance. Each workload consists of 5000 random 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 serviced.

We compare the efficiency of both workloads by measuring the average per-request *head time*. A request's head time is the amount of time that the disk head is dedicated to that request. The average head time is the reciprocal of request throughput (i.e., I/Os per second). Therefore, higher disk efficiency will result in a shorter average head time, all else being equal. We introduce head time as a metric because it allows us to identify component delays more easily.

For *onereq* requests, head 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 is the elapsed time from when a request is sent to the disk to when completion is reported. For *onereq* requests, the read/write head is idle for part of this time, because the only outstanding request is waiting for a bus transfer to complete. For *tworeq* requests, the head 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 head times for the *onereq* and *tworeq* workloads are shown graphically in Figure 5.

Read performance. Figure 6 shows the improvement given by track-aligned accesses on the Atlas 10K II. For track-sized requests, head 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 previous request's bus transfer with the current request's media transfer.

Because bus and media transfers are overlapped, the head time for a track-aligned, track-sized request in the *tworeq* workload is 8.3 ms (calculated as shown in Figure 5). Subtracting 2.2 ms average seek time from the head time yields 6.1 ms. This observed value is very close to the

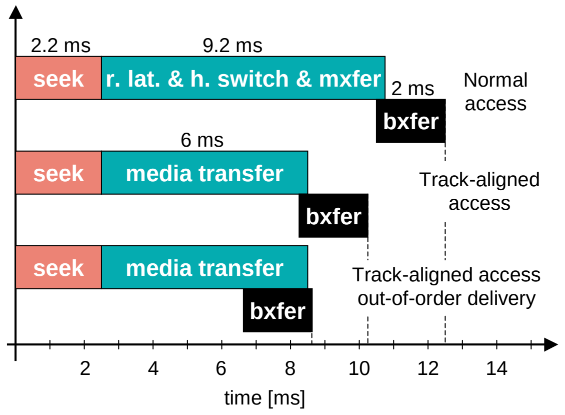


Figure 7: **Breakdown of measured response time for a zero-latency disk.** "Normal access" represents track-unaligned access, including seek, rotational latency (*r.lat.*), head switch, media transfer (*mxfer*), and bus transfer (*bxfer*). For track-aligned access, the in-order bus transfer does not overlap the media transfer. With out-of-order bus delivery, overlap of bus and media transfers is possible.

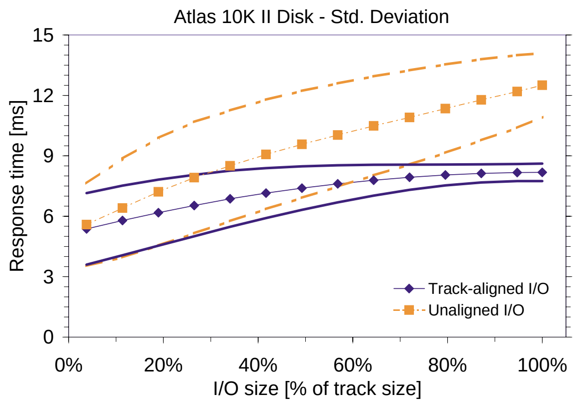


Figure 8: **Response time and its standard deviation for track-aligned and unaligned disk access.** The thin lines with markers represent the average response time, and the envelope of thick lines is the response time ± one standard deviation. The data shown in the graph was obtained by running the *onereq* workload on a simulated disk configured with an infinitely fast bus to eliminate the response time variance due to in-order bus delivery.

rotation time of 6 ms, confirming that track-aligned accesses to zero-latency disks can fetch a full track in one 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 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 7. 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 6). Although the SCSI specification allows out-of-order bus delivery using the MODIFY DATA POINTER command, we are not aware of any disks that support this operation.

Write performance. Track-alignment also makes writes more efficient. For the *onereq* workload on the Atlas 10K II, the head 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 head time is 26% (from 13.8 ms to 10.2 ms). These reductions correspond to efficiency increases of 39% and 35%, respectively.

The larger *onereq* improvement, relative to reads, occurs because the seek and bus transfer are overlapped. The disk can initiate the seek as soon as the write command arrives. While the seek is in progress, the data is transferred to the disk and buffered. Since the average seek for the *onereq* workload is 2.2 ms and the data transfer takes about 2 ms, the data usually arrives at the disk before the seek is complete and the zero-latency write begins.

Importance of zero-latency access. The head time reductions for the other zero-latency disk (the Atlas 10K) are 16% and 32% for track-sized reads in the *onereq* and *tworeq* workloads, corresponding to 19% and 47% higher efficiencies. These reductions are smaller due to the Atlas 10K's longer average seek time of 2.4 ms.

Head time does not drop as significantly for track-aligned reads on disks that do not support zero-latency access: 6% for the IBM Ultrastar 18ES and 8% for the Seagate Cheetah X15. For these disks, aligning accesses on track boundaries only eliminates the 0.8–1.1ms head switch time—the rotational latencies of 4 ms (Ultrastar) and 2 ms (Cheetah) are still incurred.

Response time variance. Track-aligned access can significantly lower the standard deviation, σ, of response time as seen in Figure 8. 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. Lower variance makes response times more predictable, allowing soft real-time applications to use tighter bounds in scheduling and thereby achieve higher utilization.

Track-based requests also have lower worst-case access times, since rotational latency and head switch time are avoided.

5.3 FFS experiments

Building on the disk-level results, this section compares our prototype traxtent-aware FFS to unmodified FFS. We also include results for a modified FFS, here called *fast start* FFS, that aggressively prefetches contiguous blocks. The unmodified FFS slowly ramps up its prefetching as it observes sequential access to a file. The fast start FFS, on the other hand, prefetches up to 32 contiguous blocks on the first access to a file, thus approximating the behavior of the traxtent-aware FFS (albeit with larger requests and no knowledge of track boundaries).

Each test is performed on a freshly-booted system with a clean partition on a Quantum Atlas 10K. The tests verify the expected performance effects: small penalty for single sequential scan, substantial benefit for interleaved scans, and no effect on small file activity. We also identify and measure the worst-case scenario. The results are summarized in Table 2.

Single large file. The first experiment is an I/O-bound linear scan through a 4 GB file. As expected, traxtent-FFS runs 5% slower than unmodified FFS or fast start FFS (199.8 s vs. 189.6 s and 188.9 s respectively). This is because FFS is optimized for large sequential single-file access and reads at the maximum disk streaming rate, whereas traxtent-FFS inserts an excluded block one out of every twenty blocks (5%). This penalty could be eliminated by changing the file system cache to support buffering of partial blocks (much like IP fragments) instead of using excluded blocks in large files; this approach would give the block-based system extent-like flexibility.

Multiple large files. 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 speedup from improved disk efficiency. For 512 MB files, traxtent-FFS completes 19% faster than unmodified FFS or fast start FFS. A more detailed analysis shows that traxtent-FFS performs 6724 I/Os (average size of 160 KB) in 56.6 s while unmodified FFS performs only 4108 I/Os (mostly 256 KB) but requires 69.7 s. The fast start FFS performs 4094 I/Os (all but one at 256 KB) and requires 70.0 s. Subtracting media transfer time, unmodified FFS incurs 6.9 ms of overhead (seek + rotational latency + track switch time) per request, and traxtent-FFS incurs only 2.2 ms of overhead per request. In fact, the 19% improvement in overall completion time corresponds to an improvement in disk efficiency of 23%, exactly matching the predicted difference between single-track accesses and 256 KB unaligned accesses on an Atlas 10K disk.

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 traxtent-FFS over unmodified FFS (124.9 s vs. 156.9 s). The fast start FFS finished in 155.3 s.

Small Files. Two application benchmarks are used to verify that the traxtent modifications do not penalize small file workloads. Postmark [21] simulates the small-file activity of busy Internet servers. Our experiments use Postmark v1.11 and its default parameters: 5–10KB files and 1:1 read-to-write and create-to-delete ratios. SSH-build [38] represents software development activity, replacing the Andrew benchmark. Its three phases unpack the compressed tar archive of SSH v1.2.27, generate the header files and Makefiles, and build the program executable.

As expected, we observe little difference. The SSH-build results differ by less than 0.2%, because the file system activity is dominated by small synchronous writes and cache hits. The fast start FFS performs exactly like the traxtent FFS having an edge of 0.2% over the unmodified FFS. Postmark is 4% faster with traxtents (55 transactions/second versus 53 for both unmodified and fast start FFS), because the few track switches are avoided. Fast start is not important for Postmark, because the files consist of only 1–3 blocks.

One might view these results as a negative indication of traxtents' value, but they are not. Recall that FreeBSD FFS does not explicitly group small files into large disk requests. Such grouping has been shown to yield 2–8$\times$ throughput increases for static web servers [20], web proxy caches [39], and software development activities [15]. Based on our measurements, we expect that the additional 50% increase in throughput from traxtents would be realized given such grouping.

Worst case scenario. As expected, we observe no penalty to small file I/O and a minimal (5%) penalty to the unoptimized single stream case. For random file I/O, FFS's "sequential count" prefetch control replaces the traxtent-based fetch mechanism, preventing useless full-track reads. The one remaining worst-case scenario would be single-block reads to the beginnings of many large files; in this case, the original FFS will fetch the first 8KB block and prefetch the second, whereas the modified FFS will fetch the entire first traxtent ($\approx$ 160 KB). To evaluate this scenario, we ran an experiment, called `head *`, that reads the first byte of 1000 200 KB files. The results show a 45% penalty for traxtents (3.6 s vs.

	4GB `scan`	512MB `diff`	1GB `copy`	Postmark	SSH-build	`head *`
unmodified	189.6 s	69.7 s	156.9 s	53 tr/s	72.0 s	4.6 s
fast start	188.9 s	70.0 s	155.3 s	53 tr/s	71.5 s	5.5 s
traxtents	199.8 s	56.6 s	124.9 s	55 tr/s	71.5 s	5.2 s

Table 2: **FreeBSD FFS results.** All but the `head *` values are an average of three runs. The individual run times deviate from their average by less than 1%. The `head *` value is an average of five runs and the individual runs deviate by less than 3.5%. Postmark reported the same number of transactions per second in all three runs for the respective FFS, except for one run of the unmodified FFS that reported 54 transactions per second.

5.2 s), closely matching the predicted per-request service time difference (5.6 ms vs. 8.0 ms). Fortunately, this scenario is not often expected to arise in practice. Not surprisingly, the fast start FFS performs even worse than the traxtent FFS with an average runtime of 5.5 s as it prefetches even more unnecessary data.

5.4 Video servers

A video server is designed to serve large numbers of video streams to clients at guaranteed rates. To accomplish this, the server first fetches one time interval of video (e.g., 0.5 s) for each stream. This set of fetches is called a *round*. Then, while the data are transferred to clients from the server's buffers, the server schedules the next round of requests. Since the per-interval disk access time is less than the round time, many concurrent streams can be supported by a single disk. Further, by spreading video streams across *D* disks, *D* times as many concurrent streams can be supported.

The per-interval disk request size, *IOsize*, represents a trade-off between throughput (the number of concurrent streams) and other considerations (buffer space and start-up latency). *IOsize* must be large enough so that achieved disk bandwidth (disk efficiency times peak bandwidth) exceeds *V* times the video bit rate, where *V* is the number of concurrent video streams supported. As *IOsize* increases, both disk efficiency and $Time_{perIO}$ increase, increasing both the number of video streams that can be supported and the round time, which is defined as *V* times $Time_{perIO}$.

Round time determines the startup latency of a newly admitted stream. Assuming the video server spreads data across *D* disks, the worst-case startup latency is round time times $(D+1)$ [34]. The buffer space required at the server is $2 \times IOsize_{\text{disk}} \times V$. In practice, *IOsize* is chosen to meet system goals given a trade-off between startup latency and the maximum number of supportable streams. Since track-aligned access increases disk efficiency, it enables more concurrent streams to be serviced at a given *IOsize*.

5.4.1 Soft real-time

Most video server projects, such as Tiger [3] and RIO [34], provide soft real-time guarantees. These systems guarantee 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.

We evaluate two video servers (one traxtent-aware and one not), each containing 10 Quantum Atlas 10K II disks, using the same approach as the RIO video server [34]. First, we measured the time to complete a given number of simultaneous, random track-sized requests. 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 each disk's 40 MB/s streaming bandwidth.)

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 array), the track-aligned system can support up to 70 streams per disk. In contrast, the unaligned system is only able to support 45 streams per disk. Thus, the track-aligned system 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. This increase in I/O size 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.

5.4.2 Hard real-time

Although many video servers implement soft real-time requirements, there are applications that require hard real-time guarantees. In their admission control algorithms, these systems must assume the worst-case response time to ensure that no deadline is missed. In

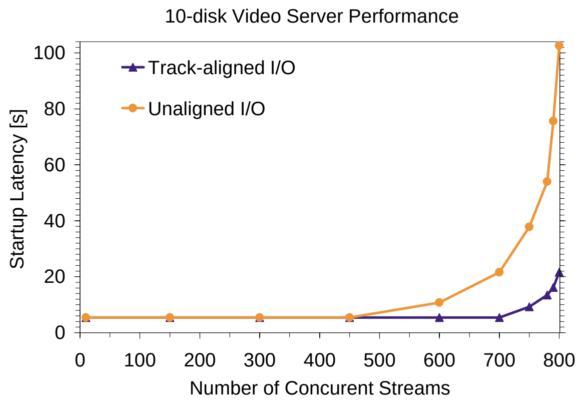


Figure 9: **Worst-case startup latency of a video stream for track-aligned and unaligned accesses.** The startup latency is shown for a 10-disk array of Quantum Atlas 10K II disks, which can support up to 800 concurrent streams.

computing the worst-case response time, one assumes the worst-case seek, transfer time, and rotational latency. Both the track-aligned and unaligned systems have the same values for the worst-case seek[2]. However, the worst-case rotational latency for unaligned access is one revolution, whereas track-based access suffers no rotational latency. The worst-case transfer time will be similar except that the unaligned system must assume at least one head switch will occur for each request. 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 access. Unaligned I/O size must exceed 2.5 MB, with a maximum startup latency of 60.5 seconds, to achieve the same efficiency as the track-aligned system.

5.5 Log-structured File System

The log-structured file system (LFS) [33] 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 [24, 37]. Additionally, the transfer of large segments hurts the performance of small synchronous reads [5, 24]. Given these conflicting pressures, the choice of segment size must balance write efficiency, cleaning cost, and small synchronous I/O performance. Matching segments to track boundaries can yield higher write efficiency with smaller segments and thus lower cleaning costs.

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. [24], which is a refinement of the *write cost* metric defined for the Sprite implementation of LFS [33]. It expresses the cost of writes in the file system, assuming that all data reads are serviced from the system cache. The *OWC* metric is defined as the product of write cost and disk transfer inefficiency:

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

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. *TransferInefficiency*, on the other hand, depends only on disk characteristics. Therefore, we can use the *WriteCost* values given by Matthews et al. for their Auspex server trace [24] and measured *TransferInefficiency* values like those in Figure 1.

Figure 10 shows that *OWC* is lower with track-aligned disk access and that 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

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

where $S_{segment}$ is the segment size (in bytes) and T_{pos} is the average positioning time (i.e., seek and rotational latency). To verify 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 10 (labeled "5.2 ms*40 MB/s") with the *OWC* values for the

[2]The worst-case time for V seeks is much smaller than V times a full strobe seek (seek from one edge of the disk to the other), decreasing with increasing number (V) of concurrent streams [31]. This is because the disk scheduler can sort the requests in each round to minimize total seek distance. The worst-case seek time charged to a stream is equal to the worst-case scheduled seek route that serves all streams divided by the number of streams.

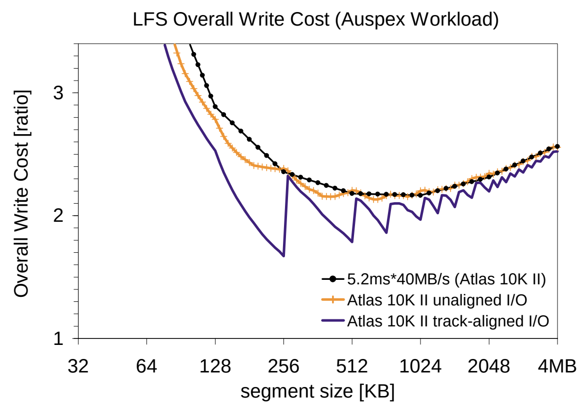


Figure 10: **LFS overall write cost for the Auspex trace as a function of segment size.** The line labeled "5.2 ms*40 MB/s" is the overall write cost predicted by the transfer inefficiency model described by Matthews et al. [24].

track-aligned and unaligned I/O. Because the empirical values are for the disk's first zone, the model values are too: 2.2 ms average seek, 3 ms average rotational latency, and peak bandwidth of 40 MB/s. As expected, the model is a good match for the unaligned case.

5.5.1 Variable segment size

As shown in Figure 10, the lowest write cost is achieved when the size of a segment matches the size of a track. However, different tracks may hold different numbers of LBNs. Therefore, an LFS must allow for variable segment sizes in order to match segment boundaries to track boundaries. Fortunately, doing so is straightforward.

In an LFS, the segment usage table records information about each segment. In the SpriteLFS implementation [33], 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 [36] stores this table in a special file called the IFILE. Because of its frequent use, this file is almost always in the file system's cache.

Variable-sized segments can be supported by augmenting the per-segment information in the segment usage table with a starting location (the LBN) and length. During the initialization, each segment's starting location and length are set according to the corresponding 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.

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 outer 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 [27] 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 [8, 10, 13, 46, 49]. For small writes, these systems detect the position of the head and re-map data to the nearest free block in order to minimize the positioning costs [10, 46]. For small reads, the SR-Array [49] 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.

Acknowledgements

Bill Nace contributed to our initial investigation into track-aligned extents. We thank our shepherd John Wilkes and the anonymous reviewers for helping us refine this paper. 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. We thank IBM and Intel for hardware grants supporting our research effots. John Griffin is funded in part by a National Science Foundation Graduate Fellowship.

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] Chia Chao, Robert English, David Jacobson, Alexander Stepanov, and John Wilkes. *Mime: high performance parallel storage device with strong recovery guarantees*. Technical report HPL-92-9. 18 March 1992.

[9] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, **26**(2):145–185, June 1994.

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

[11] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh. The Logical Disk: a new approach to improving file systems. *ACM Symposium on Operating System Principles* (Asheville, NC), pages 15–28, 5–8 December 1993.

[12] 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.

[13] 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.

[14] 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.

[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, Bruce L. Worthington, and Yale N. Patt. *The DiskSim simulation environment version 1.0 reference manual*, Technical report CSE–TR–358–98. Department of Computer Science and Engineering, University of Michigan, February 1998.

[17] 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.

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

[19] David Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. *Winter USENIX Technical Conference* (San Francisco, CA, 17–21 January 1994), pages 235–246. USENIX Association, 1994.

[20] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. *ACM SIGOPS. European workshop: Systems support for worldwide applications* (Connemara, Ireland, September 1996), pages 141–148. ACM, 1996.

[21] Jeffrey Katcher. *PostMark: a new file system benchmark*. Technical report TR3022. Network Appliance, October 1997.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

[29] 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.

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

[31] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. *International Multimedia Conference* (Anaheim, CA, 01–06 August 1993), pages 225–234. ACM Press, 1993.

[32] ReiserFS. http://www.namesys.com/.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems. *USENIX Annual Technical Conference* (San Diego, CA), 18–23 June 2000.

[39] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein. Storage management for web proxies. *Annual USENIX Technical Conference* (Boston, MA, 25–30 June 2001), pages 203–216, 2001.

[40] 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.

[41] 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.

[42] Iceberg 9200 disk array subsystem. Storage Technology Corporation, 2270 South 88th Street, Louisville, CO 80028-4358, 9 June 1995.

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

[44] 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.

[45] 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.

[46] 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, 1999.

[47] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, **14**(1):108–136, February 1996.

[48] 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.

[49] 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.