A Linux Implementation Validation of Track-Aligned Extents and Track-Aligned RAIDs

Jin Qian, Christopher Meyers, and An-I Andy Wang
Florida State University, {qian, meyers, awang@cs.fsu.edu}

Abstract

Through clean-slate implementation of two storage optimizations—track-aligned extents and track-aligned RAIDs—this paper shows the values of independent validations. The experience revealed many unanticipated disk and storage data path behaviors as potential roadblocks for wide deployment of these optimizations, and also identified implementation issues to retrofit these concepts to legacy data paths.

1 Introduction

Validation studies are common in science, but less emphasized in computer science, because a rapidly moving field tends to focus on advancing the frontier.

Through a clean-slate Linux implementation of two storage optimization techniques, we aim to demonstrate the values of validations. (1) Existing validations are often implicit when the original contributors extend their work. Therefore, subtle assumptions on the OS platforms, system configurations, and hardware constraints can become obscure over time. Independent validations help identify these roadblocks, to ease the technology transfer for wide adoptions. (2) Independent validations can explore design alternatives to verify the resiliency of a concept to different platforms and hardware generations.

This paper presents a validation study of track-aligned extents [9] and track-aligned RAIDs [10]. Both showed significant performance gains. Our experience shows many unanticipated disk features and interactions along the storage data path, and identifies implementation issues to retrofit these concepts to the legacy data path.

2 Track-aligned Extents

The basic idea of track-aligned extents is that an OS typically accesses disks in blocks, each containing multiple sectors. Therefore, accessing a block can potentially cross a track boundary and incur additional head positioning time to switch tracks. By exploiting track boundaries, the performance of accessing a track size of data can improve substantially [9].

2.1 Original Implementation

Track-aligned extents [9] was built under FreeBSD by modifying FFS [7]. Two methods were proposed to extract disk track boundaries, one from the user space and one via SCSI commands. The track boundaries are extracted once, stored, and imported to FFS at mount times. The FFS free block bitmaps are modified to exclude blocks that cross track boundaries. The FFS prefetching mechanism was modified to stop at track boundaries, so that speculative disk I/Os made for sequential accesses would respect track alignments.

Track-aligned extents rely on disks that support zero-latency access, which allows the tail-end of a requested track to be accessed before the beginning of the requested track content [13]. This feature allows an aligned track of data to be transferred without rotational overhead.

With Quantum Atlas 10K II disks, the measured results showed 50% improvement in read efficiency. Simulated and computed results also demonstrated improved disk response times and support for 56% higher concurrency under video-on-demand workloads.

2.2 Recreating Track-aligned Extents

Recreating track-aligned extents involves (1) finding the track boundaries and the zero-latency access disk characteristics, (2) making use of such information, and (3) verifying its benefits. The hardware and software experimental settings are summarized in Table 1.

Hardware/software	Configurations
Processor	Pentium D 830, 3GHz, 16KB L1 cache, 2x1MB L2 cache
Memory	128 MB or 2GB
RAID controller	Adaptec 4805SAS
Disks tested	Maxtor SCSI 10K5 Atlas, 73GB, 10K RPM, 8MB on-disk cache [6] Seagate CheetahR 15K.4 Ultra320 SCSI, 36GB, 8MB on-disk cache [12] Fujitsu MAP3367NC, 10K RPM, 37GB, with 8MB on-disk cache [5]
Operating system	Linux 2.6.16.9
File system	Ext2 [4]

Table 1: Experimental settings.

2.3 Extracting Disk Characteristics

User-level scanning: Since the reported performance gains for track alignments are high, conceivably a user-level program can observe timing variations to identify track boundaries. A program can incrementally issue reads, requesting one more sector than before, starting from the 0^{th} sector. As the request size grows, the disk bandwidth should first increase and then drop as the request size exceeds the size of the first track (due to track switching overhead). The process can then repeat, starting from the first sector of the previously found track. Binary search can improve the scheme.

To reduce disturbances caused by various disk data path components, we used the `DIRECT_IO` flag to

bypass the Linux page cache, and we accessed the disk as a raw device to bypass the file system. We used a modified `aacraid` driver code to bypass the SCSI controller, and we used `sdparm` to disable the read cache (`RCD=1`) and prefetch (`DPTL=0`) of the disk.

As a sanity check, we repeated this experiment with an arbitrary starting position at the 256th sector (instead of the 0th sector). Additionally, we repeated this experiment with a random starting sector between 0 and 512, with each succeeding request size increasing by 1 sector (512 bytes).

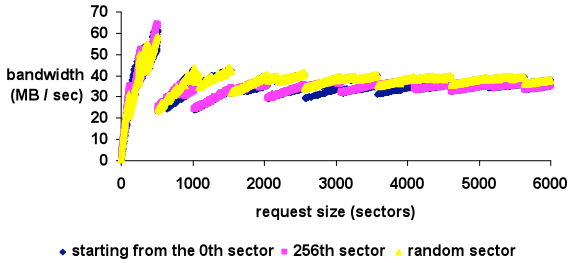

Figure 1: Bandwidth for various read request sizes from varying starting sectors on a Maxtor disk.

Surprisingly, although Figure 1 exhibits bandwidth "cliffs," the characteristic trends are not sensitive to the starting location of requests, suggesting that those cliffs are caused by sources of data misalignments other than tracks. Some possibilities are transfer granularity of the DMA and the management granularity of IO buffers. The graph also suggests the presence of other optimizations that are not disabled. For example, the high bandwidth before the first cliff far exceeds our expected performance gain. [2] conjectures that the DEC prefetch scheme implemented in Maxtor may override specified disk settings at times and proceed with prefetching. Additionally, for certain ranges of request sizes (e.g., between 1,000 and 1,500 sectors), the average bandwidth shows multimodal behaviors.

To verify that those cliff locations are not track boundaries, we wrote a program to access random cliff locations with the access size of 512 sectors (256KB), as indicated by the first cliff location. We ran multiple instances of this program concurrently and perceived no noticeable performance difference compared to the cases where the accesses started with random sectors.

SCSI diagnostic commands: Unable to extract track boundaries from a naive user-level program, we resorted to SCSI `SEND/RECEIVE DIAGNOSTIC` commands to map a logical block address (LBA) to a physical track, surface, and sector number.[1] However, this translation for large drives is very slow, and it took days to analyze a 73-GB drive. We modified the `sg_senddiag` program in the Linux `sg3_utils` package to speed up the extraction process, according to the following pseudocode:

1. Extract from LBA 0 sector-by-sector until either track number or surface number changes. Record LBA and the physical address of this track boundary. Store the track size S.
2. Add S to the last known track boundary T and translate S + T and S + T – 1.
   a. If we detect a track change between S + T and S + T – 1, then S + T is a new boundary. Record the boundary. Go to step 2.
   b. If there is no change between S + T and S + T – 1, the track size has changed. Extract sector-by-sector from the previous boundary until we detect a new track boundary. Record the boundary, update S, and go to step 2.
3. If sector reaches the end of the disk in step 2, exit.

Through this scheme, we extracted the layout mapping specifics that are not always published in vendors' datasheets and manuals [5, 6, 12] in about 7 minutes.

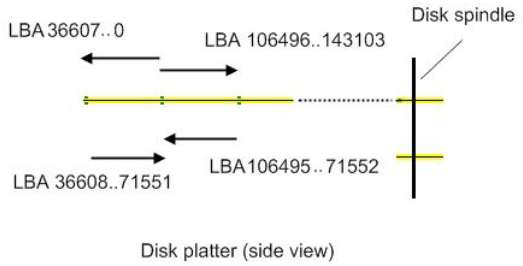

Figure 2: Non-monotonic mapping between LBA and track numbers.

First, the LBA mapping to the physical track number is not monotonic (Figure 2). For the Maxtor drive, LBA 0 starts on track 31 of the top surface and increases outward (from the disk spindle) to track 0, and then the LBA continues from the bottom surface of track 0 inward to track 31. Next, the LBA jumps to track 63 of the bottom surface growing outward to track 32, and then switches back to the top surface's track 32 and continues inward to track 63. The pattern repeats.

Variants of this serpentine numbering scheme [1, 11] are observed in Seagate [12] and Fujitsu [5] drives as well. At the first glace, one might conjecture this numbering scheme relates to the elevator and scanning-based IO schedulers, but this scheme is attributed to the faster timing when switching a head track-to-track on the same surface than when switching to a head on a different surface [11].

Second, the track size differs even for the same disk model from the same vendor, due to the manufacturing process of the disks. After assembly, the disk head frequency response is tested. Disk heads with a lower frequency response are formatted with fewer sectors per track [2]. We purchased 6 Maxtor 10K V drives at the same time and found 4 different LBA numbering schemes (Table 2). The implication is that track extraction needs to be performed on every

[1] We did not use DIXtrac [8] for the purpose of clean-slate implementation and validation.

disk, even those from the same model. Track size may differ in the same zone on the same surface due to defects. Thus, we are no longer able to calculate the track boundary with zone information but have to extract all tracks.

Serial number	Surface 0, outer most track	Surface 1, outer most track
J20 Q3 CZK	1144 sectors	1092 sectors
J20 Q3 C0K/J20 Q3 C9K	1092 sectors	1144 sectors
J20 TK 7GK	1025 sectors	1196 sectors
J20 TF S0K/J20 TF MKK	1060 sectors	1170 sectors

Table 2: Track sizes of Maxtor 10K V drives.

Verifying track boundaries: To verify track boundaries, we wrote a program to measure the elapsed time to access 64 sectors with shifting offsets from random track boundaries. The use of 64 sectors eases the visual identifications of boundaries. We measured tracks only from the top surface within the first zone of a Maxtor disk, so we could simplify our experiment by accessing a mostly uniform track size of 1,144 sectors.

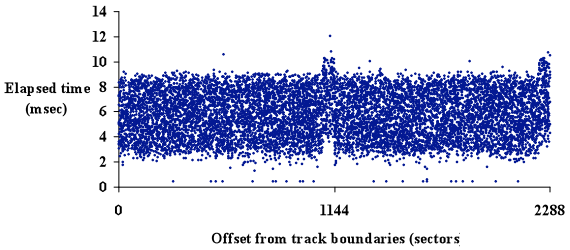

Figure 3: Elapsed time to access 64 sectors, starting from different offsets from various track boundaries on a Maxtor drive (the track size is 1,144 sectors).

Figure 3 confirms extracted track boundaries. Each data point represents the time to access a 64-sector request starting from a randomly chosen sector offset from a track boundary. The 6-msec timing variation reflects the rotation delay for a 10,000 RPM drive. The average elapsed time for accessing 64 sectors across a track boundary is 7.3 msec, compared to 5.7 msec for not crossing the track boundaries. Interestingly, the difference of 1.6 msec is much higher than the track switching time of 0.3 to 0.5 msec [6]. We also verified this extraction method with other vendor drives. The findings were largely consistent.

Zero-latency feature verification: Since the effectiveness of track-aligned extents relies on whether a disk can access the data within a track out-of-order, we performed the tests suggested in [13]. Basically, we randomly picked two consecutive sectors, read those sectors in reverse LBA order, and observed the timing characteristics. We performed the test with various caching options on.

As shown in Figure 4, with a Maxtor drive, 50% of the time the second request is served from the on-disk cache, indicating the zero-latency capability. (We did not observe correlations between the chosen sectors and whether the zero-latency feature is triggered.) In contrast, the other two drives always need to wait for a 3- to 6-msec rotational delay before serving the second sector request. For the remainder of the paper, we will use the Maxtor drives.

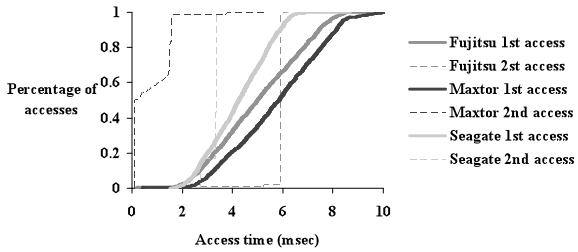

Figure 4: CDF of disk access times for accessing random sets of two consecutive LBAs in the reverse order.

2.4 Exploiting Track Boundaries

The track boundary information can be exploited at different levels.

User level: One possibility is to create a user program to make use of this track information. Similar to the disk defragmentation, instead of moving file blocks to reduce the level of fragmentation, we can move blocks to align with track boundaries. This approach avoids kernel changes and can make files smaller than a track not cross track boundaries, and files larger than a track aligned to track boundaries.

However, this approach needs to overcome many tricky design points. For example, certain blocks are referenced from many places (e.g., hardlinks). Moving those blocks requires tracking down and updating all references to the block being moved. Such information might not be readily available.

File system level: We can mark certain sectors as bad so a file system cannot allocate blocks that consist of sectors across track boundaries. However, this method does not prevent a track-size file from being allocated across two tracks. This approach also anticipates some bandwidth loss when a single IO stream accesses multi-track files due to unused sectors. However, when a system is under multiple concurrent IO streams, the performance benefits of accessing fewer tracks when multiplexing among streams can outweigh the performance loss.

Implementation: We implemented track-aligned extents in ext2 [4] under Linux. First, we used the track boundary list extracted by the SCSI diagnostic commands as the bad-block list input for the `mke2fs` program, which marks all of these blocks, so that they will not be allocated to files. We also put this list in a kernel module along with two functions. One initializes and reads the list from user space. The other is used by different kernel components to find a track boundary after a given position.

We then modified the ext2 pre-allocation routine to allocate in tracks (or up to a track boundary). One disadvantage of this approach is over-allocation, but the unused space can later be returned to the system. However, should the system anticipate mostly track-size accesses, we are less concerned with the wasted space. For instance, database and multimedia applications can adjust their access granularity accordingly. With the aid of this list, we can also change the read-ahead to perform prefetches with respect to track boundaries.

Our experience suggests that individual file systems only need to make minor changes to benefit from track alignments.

2.5 Verification of the Performance Benefits

We used the sequential read and write phases of the Bonnie benchmark [3], which is unaware of the track alignments. The write phase creates a 1-GB file, which exceeds our 128-MB memory limit. We enabled SCSI cache, disk caching, and prefetch to reflect normal usage. Each experiment was repeated 10 times, analyzed at a 90% confidence interval.

Figure 5 shows the expected 3% slowdown for a single stream of sequential disk accesses, where skipped blocks that cross track boundaries can no longer contribute to the bandwidth.

We also ran `diff` from GNU `diffutils` 2.8.1 to compare two 512-MB large files via interleaved reads between two files, with the `-speed-large-files` option. Without this option, `diff` will try to read one entire file into the memory and then the other file and compare them if memory permits, which nullifies our intent of testing interleaved reads. Figure 6 shows that track-aligned accesses are almost twice as fast as the normal case. In addition, we observed that disk firmware prefetch has no regard for track boundaries. Disabling on-disk prefetch further speeds up track-aligned access by another 8%. Therefore, for subsequent experiments, we disabled disk firmware prefetch for track-aligned accesses.

Additionally, we conducted an experiment that involves concurrent processes issuing multimedia-like traffic streams at around 500KB/sec. We used 2GB for our memory size. We wrote a script that increases the number of streams by one after each second, and the script records the startup latency of each new stream. Each emulated multimedia streaming process first randomly selects a disk position and sequentially accesses the subsequent blocks at the specified streaming rate. We assumed that the acceptable startup latency is around 3 seconds, and the program terminates once the latency reaches 3 seconds.

Figure 7 shows that the original disk can support up to 130 streams with a startup latency within 3 seconds. A track-size readahead window can reduce the latency at 130 streams by 30%, while track-aligned access can reduce the latency by 55%.

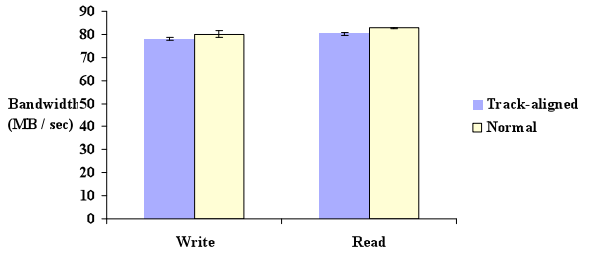

Figure 5: Bandwidth comparisons between conventional and track-aligned accesses to a single disk, when running the Bonnie benchmark.

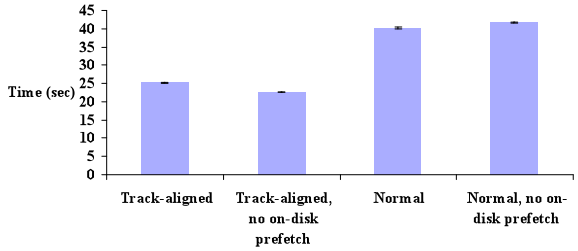

Figure 6: Speed comparisons between conventional and track-aligned accesses to a single disk, `diff`ing two 512MB files with 128MB of RAM.

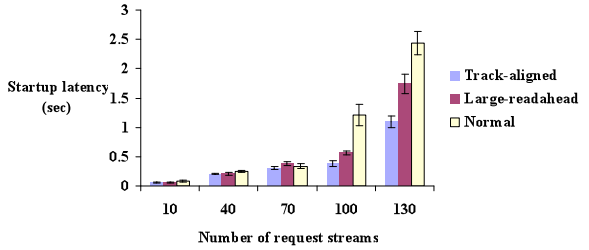

Figure 7: Startup latency comparisons of conventional I/O requests, requests with a one-track prefetch window, and track-aligned requests on a single disk, with a varying number of multimedia-like request streams.

3 Track-aligned RAIDs

Original implementation: Schindler et al [10] proposed Atropos, a track-aligned RAID. The implementation was through a user-level logical volume manager process. The process bypasses conventional storage data paths and issues raw IOs. An application needs to be linked with a stub library to issue reads and writes. The library uses shared memory to avoid data copies and communicates with Atropos through a socket.

Without the conventional storage data path, Atropos is responsible for scheduling requests with the help of a detailed disk model. Atropos also needs to duplicate logics provided by conventional RAID levels. As a proof of concept, the measured prototype implemented RAID-0 (no redundancy) and RAID-1

(mirroring), although issues relevant to other RAID levels are addressed in the design.

To handle different track sizes due to disk defects, for simplicity Atropos skips tracks that contain more than a threshold number of defects, which translates to about 5% of storage overhead.

The performance for track-aligned RAIDs matches the efficiency expectation of track-aligned extents.

Recreating Track-aligned RAIDs: Our clean-slate validation implements track-aligned RAIDs via modifying RAID-5 (distributed parity), retrofitting the conventional storage data path. Thus, unmodified applications can enjoy the performance benefit as well. However, we had to overcome a number of implementation constraints.

Recall from Section 2.3 that the track sizes can differ even from the same disk model. This difference was much more than that caused by defects. Therefore, we need measures beyond skipping tracks. For one, we can construct stripes with tracks of different sizes. Although this scheme can work with RAID-0, it does not balance load well or work well with other RAID levels. For example, RAID-5 parity is generated via XORing chunks (units of data striping) of the same size. Suppose we want the chunk unit to be set to the size of a track. If we use the largest track size as the chunk unit, some disks need to use 1+ tracks to form a chunk. Or we can use the smallest track size as the chunk unit, leading to about 10% of unused sectors for disks with larger track sizes.

Additionally, we observed that parity in RAIDs can interact poorly with prefetching in the following way. Take RAID-5 as an example. At the file system level, prefetching one track from each non-parity disk involves a prefetching window that is the size of a track multiplied by the number of disks that do not contain the parity information. However, as a RAID redirects the contiguous prefetching requests from the file system level, the actual forwarded track-size prefetching requests to individual disks are fragmented, since reads in RAIDs do not need to access the parity information.

Another poor interaction is the Linux plug and unplug mechanisms associated with disk queues and multi-device queues. These mechanisms are designed to increase the opportunities for data reordering by introducing artificial forwarding delays at times (e.g., 3 msec), and do not respect track boundaries. Therefore, by making these mechanisms aware of track boundaries, we were finally able to make individual disks in a RAID-5 access in a track-aligned manner.

Implementation: We modified Linux software RAID-5 to implement the track-aligned accesses. We altered the `make_request` function, which is responsible for translating the RAID virtual disk address into individual disk addresses. If the translated requests crossed track boundaries, the unplug functions for individual disk queues were explicitly invoked to issue track-aligned requests.

To prevent the parity mechanisms from fragmenting track-size prefetching requests, we modified RAID-5. Whenever the parity holding disk in a stripe was the only one not requested for that stripe, we filled in the read request for that disk and passed it down with all others. When this dummy request was completed, we simply discarded the data. The data buffer in Linux software RAID-5 is pre-allocated at initialization, so this implementation does not cause additional memory overhead.

Verification of performance benefits: We compared the base case RAID-5 with a track-aligned RAID-5 with five disks, and a chunk size of 4KB. For the Bonnie benchmark, we used a 1-GB working set with 128MB of RAM. Figure 8 shows that the write bandwidth for the three system settings falls within a similar range due to buffered writes. However, for read bandwidth, the track-aligned RAID-5 outperforms the conventional one by 57%.

The `diff` experiment compared two 512-MB files with 128MB of RAM. Figure 9 shows that the track-aligned RAID-5 can achieve a 3x factor speedup compared to the original RAID-5.

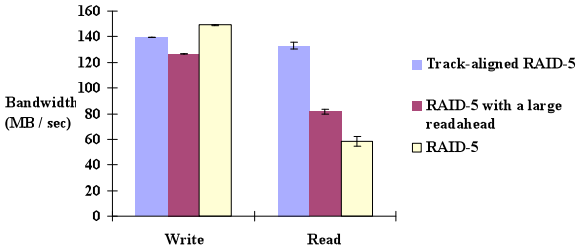

Figure 8: Bandwidth comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of four tracks, and the original RAID-5, running Bonnie with 1GB working set and 128MB of RAM.

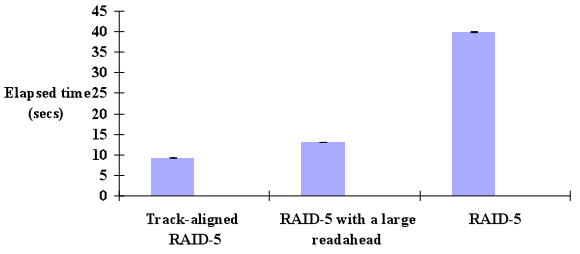

Figure 9: Elapsed time comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of four tracks, and the original RAID-5, when running `diff` comparing two 512MB files.

For the multimedia-like workload with 2GB of RAM, the track-aligned RAID-5 demonstrates a 3.3x better scaling in concurrency than the conventional RAID-5 (Figure 10), where a RAID-5 with a readahead window comparable to the track-aligned RAID-5 contributes only less than half of the scaling improvement. The latency improvement of track-aligned RAID-5 is

impressive considering that the RAID-5 was expected to degrade in latency when compared to the single-disk case, due to the need to wait for the slowest disk for striped requests. Track-aligned accesses reduce the worst-case rotational timing variance and can realize more benefits of parallelism.

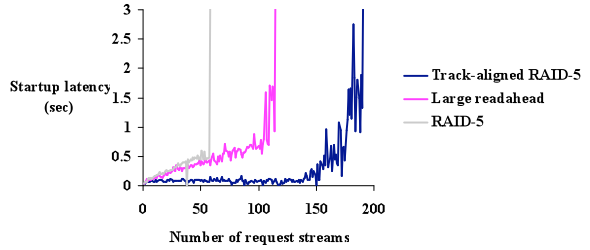

Figure 10: Startup latency comparisons of the track-aligned RAID-5, a RAID-5 with a prefetch window of four tracks, and the original RAID-5, with a varying number of multimedia-like request streams.

4 Lessons Learned and Conclusions

Through clean-slate implementations of track-aligned extents and track-aligned RAIDs, we have demonstrated important values of independent validations. First, the validation of research results obtained five years ago shows the relative resiliency and applicability of these concepts to different platforms and generations of disks. On the other hand, as the behaviors of disks and the legacy storage data path become increasingly complex, extracting physical disk geometries will likely become increasingly more difficult. Also, as disks become less homogeneous even within the same model, techniques such as track-aligned RAIDs need to devise additional measures to prevent a RAID from being limited by the slowest disk.

Second, through exploring design and implementation alternatives, we revealed many unanticipated interactions among layers of data path optimizations. On-disk prefetching, IO scheduling and aggregation, RAID parity, file system allocation, and file system prefetching—all have side effects on IO access alignment and profound performance implications. Unfortunately, the interfaces among data path layers are lacking in expressiveness and control, leading to modifications of many locations to retrofit the concepts of access alignment into the legacy storage data path, the remedy for which is another fruitful area of research to explore.

Acknowledgements

We thank Mark Stanovich and Adaptec for helping us bypass some RAID controller features. We also thank Peter Reiher and Geoff Kuenning for reviewing this paper. This research is sponsored by NSF CNS-0410896 and CNS-0509131. Opinions, findings, and conclusions or recommendations expressed in this document do not necessarily reflect the views of the NSF, FSU, or the U.S. government.

References

[1] Anderson D. You Don't Know Jack about Disks. *Storage*. 1(4), 2003.

[2] Anonymous Reviewer, reviewer comments, *the 6th USENIX Conf. on File and Storage Technologies*, 2007.

[3] Bray T. Bonnie benchmark. http://www.textuality.com/bonnie/download.html, 1996.

[4] Card R, Ts'o T, Tweedie S. Design and Implementation of the Second Extended Filesystem. *The HyperNews Linux KHG Discussion*. http://www.linuxdoc.org, 1999.

[5] Fujitsu MAP3147NC/NP MAP3735NC/NP MAP3367NC/NP Disk Drives Product/Maintenance Manual. http://www.fujitsu.com/downloads/COMP/fcpa/hdd/discontinued/map-10k-rpm_prod-manual.pdf, 2007.

[6] Maxtor Atlas 10K V Ultra320 SCSI Hard Drive. http://www.darklab.rutgers.edu/MERCURY/t15/disk.pdf, 2004.

[7] McKusick MK, Joy WN, Leffler SJ, Fabry RS. A Fast File System for UNIX, *Computer Systems*, 2(3), pp. 181-197, 1984.

[8] Schindler J, Ganger GR. Automated Disk Drive Characterization. CMU SCS Technical Report CMU-CS-99-176, December 1999.

[9] Schindler J, Griffin JL, Lumb CR, Ganger GR. Track-Aligned Extents: Matching Access Patterns to Disk Drive Characteristics. *Proc. of the 1st USENIX Conf. on File and Storage Technologies*, 2002.

[10] Schindler J, Schlosser SW, Shao M, Ailamaki A, Ganger GR. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. *Proc. of the 3rd USENIX Conf. on File and Storage Technologies*, 2004.

[11] Schlosser SW, Schindler J, Papadomanolakis S, Shao M, Ailamaki A, Faloutsos C, Ganger GR. On Multidimensional Data and Modern Disks. *Proc. of the 4th USENIX Conf. on File and Storage Technology*, 2005.

[12] Seagate Product Manual: CheetahR 15K.4 SCSI. http://www.seagate.com/staticfiles/support/disc/manuals/enterprise/cheetah/15K.4/SCSI/100220456d.pdf, 2007.

[13] Worthington BL, Ganger GR, Patt YN, Wilkes J. On-line Extraction of SCSI Disk Drive Parameters. *ACM Sigmetrics*, 1