Virtual Log Based File Systems for a Programmable Disk*

Randolph Y. Wang† Thomas E. Anderson‡ David A. Patterson†

Abstract

In this paper, we study how to minimize the latency of small synchronous writes to disks. The basic approach is to write to free sectors that are near the current disk head location by leveraging the embedded processor core inside the disk. We develop a number of analytical models to demonstrate the performance potential of this approach. We then present the design of a *virtual log*, a log whose entries are not physically contiguous, and a variation of the log-structured file system based on this approach. The virtual log based file systems can efficiently support small synchronous writes without extra hardware support while retaining the advantages of LFS including its potential to support transactional semantics. We compare our approach against traditional update-in-place and logging systems by modifying the Solaris kernel to serve as a simulation engine. Our evaluations show that random synchronous updates on an unmodified UFS execute up to an order of magnitude faster on a virtual log than on a conventional disk. The virtual log can also significantly improve LFS in cases where delaying small writes is not an option or online cleaning would degrade performance. If the current trends of disk technology continue, we expect the performance advantage of this approach to become even more pronounced in the future.

1 Introduction

In this paper, we set out to answer a simple question: how do we minimize the latency of small synchronous writes to disk?

The performance of small synchronous disk writes impacts the performance of important applications such as recoverable virtual memory [27], persistent object stores [2, 18], and database applications [33, 34]. These systems have become more complex in order to deal with the increasing relative cost of small writes [19].

Similarly, most existing file systems are carefully structured to avoid small synchronous disk writes. UFS by default delays data writes to disk. It is also possible to delay metadata writes if they are carefully ordered [9]. The Log-structured File System (LFS) [25] batches small writes. While the structural integrity of these file systems can be maintained, none of them allows small synchronous writes to be supported efficiently. Write-ahead logging systems [4, 6, 12, 32] accumulate small updates in a log and replay the modifications later by updating in place. Databases often place the log on a separate disk to avoid having the small updates to the log conflict with reads. Our interest is in the limits to small write performance for a single disk.

Because of the limitations imposed by disks, non-volatile RAM (NVRAM) or an uninterruptable power supply (UPS) is often used to provide fast stable writes [3, 14, 15, 19]. However, when write locality exceeds buffer capacity, performance degrades. There are also applications that demand stricter guarantees of reliability and integrity than that of either NVRAM or UPS. Fast small disk writes can provide a cost effective complement to NVRAM.

Our basic approach is to write to a disk location that is close to the head location. We call this *eager writing*. Eager writing requires the file system to be aware of the precise disk head location and disk geometry. One way to satisfy this requirement is to enhance the disk interface to the host so that the host file system can have precise knowledge of the disk state. A second solution is to migrate into the disk some of the file system responsibilities that are traditionally executed on the host. In the rest of this paper, we will assume this second approach, although our techniques do not necessarily depend

*This work was supported in part by the Defense Advanced Research Projects Agency (DABT63-96-C-0056), the National Science Foundation (CDA 9401156), California MICRO, the AT&T Foundation, Digital Equipment Corporation, Hewlett Packard, IBM, Intel, Sun Microsystems, and Xerox Corporation. Anderson was also supported by a National Science Foundation Presidential Faculty Fellowship.

†Computer Science Division, University of California, Berkeley, {rywang,pattrsn}@cs.berkeley.edu

‡Department of Computer Science and Engineering, University of Washington, Seattle, tom@cs.washington.edu

on the ability to run file systems inside disks.

Several technology trends have simultaneously enabled and necessitated the approach of migrating file system responsibility into the disk. First, Moore's Law has driven down the relative cost of CPU power to disk bandwidth, enabling powerful systems to be embedded on disk devices [1]. As this trend continues, it will soon be possible to run the entire file system on the disk. Second, growing at 40% per year [11], disk bandwidth has been scaling faster than other aspects of the disk system. I/O bus performance has been scaling less quickly [21]. The ability of the file system to communicate with the disk (to reorganize the disk, for example) without consuming valuable I/O bus bandwidth has become increasingly important. Disk latency improves even more slowly (at an annual rate of 10% in the past decade [21]). A file system whose small write latency is largely decided by the disk bandwidth instead of any other parameters will continue to perform well. Third, the increasing complexity of the modern disk drives and the fast product cycles make it increasingly difficult for operating system vendors to incorporate useful device heuristics into their file systems to improve performance. By running file system code inside the disk, we can combine the precise knowledge of the file system semantics and detailed disk mechanism to perform optimizations that are otherwise impossible.

The basic concept of performing writes near the disk head position is by no means a new one [5, 8, 10, 13, 23]. But these systems either do not guarantee atomic writes, have poor failure recovery times, or require NVRAM. In this work, we present the design of a *virtual log*, a logging strategy based on eager writing with these unusual features:

- Virtual logging supports fast, synchronous, and atomic disk writes without special hardware support; it serves as a base mechanism upon which efficient transactions can be built.
- The virtual log allows space occupied by obsolete entries to be reused without recopying live entries.
- The virtual log boot straps its recovery from the log tail pointer, which can be stored on disk as part of the firmware power down sequence, allowing efficient normal operations.

We discuss two designs in which the virtual log can be used to improve file system performance. The first is to use it to implement a logical disk interface. This design, called a Virtual Log Disk (VLD), does not alter the existing disk interface and can deliver the performance advantage of eager writing to an unmodified file system. In the second approach, which we have not implemented, we seek a tighter integration of the virtual log into the file system; we present the design of a variation of LFS, called VLFS. We develop analytical models and algorithms to answer a number of fundamental questions about eager writing:

- What is the theoretical limit of its performance?
- How can we ensure open space under the disk head?
- How does this approach fare as different parts of the disk mechanism improve at different rates?

We evaluate our approach against update-in-place and logging by modifying the Solaris kernel to serve as a simulation engine. Our evaluations show that an *unmodified* UFS on an eager writing disk runs about ten times faster than an update-in-place system for small synchronous random updates. Eager writing's economical use of bandwidth also allows it to significantly improve LFS in cases where delaying small writes is not an option or online cleaning would degrade performance. The performance advantage of eager writing should become more profound in the future as technology improves.

Of course, like LFS, these benefits may come at a price of potentially reducing read performance as data may not be optimally placed for future reads. But as increasingly large file caches are employed, modern file systems such as the Network Appliance file system report predominantly write traffic [15]. Large caches also provide opportunity for read reorganization before the reads happen [22].

Although the virtual log shows significant performance promise, this paper remains a preliminary study. A full evaluation would require answers to algorithmic and policy questions of the data reorganizer as part of a complete VLFS implementation. These issues, as well as questions such as how to extend the virtual logging technique to multiple disks, are subjects of our ongoing research.

The remainder of the paper is organized as follows. Section 2 presents the eager writing analytical models. Section 3 presents the design of the virtual log and the virtual log based LFS. Section 4 describes the experimental platform that is used to evaluate the update-in-place, logging, and eager writing strategies. Section 5 shows the experimental results. Section 6 describes some of the related work. Section 7 concludes.

2 Limits to Low Latency Writes

The principle of writing data near the current disk head position is most effective when the head is always on a free sector. This is not always possible. In this section, we develop a number of analytical models to estimate the amount of time needed to locate free sectors under various utilizations. These models will help us evaluate whether eager writing is a sound strategy, set performance targets for real implementations, and predict future improvements as disks improve. We also use the models to motivate new file system allocation and reorganization algorithms. Because we are interested in the theoretical limits of eager writing latency, the models are for the smallest addressable unit: a disk sector (although the validity of the formulas do not depend on the sector size).

2.1 A Single Track Model

Suppose a disk track contains n sectors, its free space percentage is p, and the free space is randomly distributed. The average number of sectors the disk head must skip before arriving at any free sector is:

$$\frac{(1-p)n}{1+pn} \quad (1)$$

Appendix A.1 shows a proof of (1) and extends it for situations where the file system block size does not equal the size of the smallest addressable disk unit.

Formula (1) is roughly the ratio between occupied sectors and free ones. This is a promising result for the eager writing approach because, for example, even at a relatively high utilization of 80%, we can expect to incur only a four-sector rotational delay to locate a free sector. For today's disks, this translates to less than 100 μs. In six to seven years, this delay should improve by another order of magnitude because it scales with platter bandwidth. In contrast, it is difficult for an update-in-place system to avoid at least a half-rotation delay. Today, this is, at best, 3 ms, and it improves slowly. This difference is the fundamental reason why eager writing can outperform update-in-place.

2.2 A Single Cylinder Model

We now extend (1) to cover a single cylinder. We compare the time needed to locate the nearest sector in the current track against the time needed to find one in other tracks in the same cylinder and take the minimum of the two. Therefore the expected latency can be expressed as:

$$\sum_x \sum_y \min(x,y) \cdot f_x(p,x) \cdot f_y(p,y) \quad (2)$$

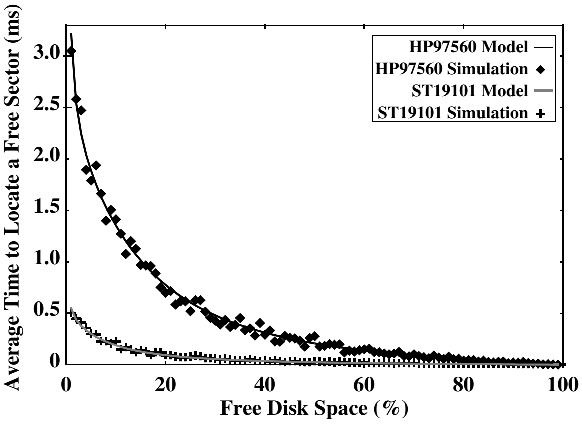

Figure 1: Amount of time to locate the first free sector as a function of the disk utilization.

	HP97560	ST19101
Sectors per Track (n)	72	256
Tracks per Cylinder (t)	19	16
Head Switch (s)	2.5 ms	0.5 ms
Minimum Seek	3.6 ms	0.5 ms
Rotation Speed (RPM)	4002	10000
SCSI Overhead (o)	2.3 ms	0.1 ms

Table 1: Parameters of the HP97560 and the Seagate ST19101 disks.

where x is the delay (in units of sectors) experienced to locate the closest sector in the current track, y is the delay experienced to locate the closest sector in other tracks in the current cylinder, and $f_x(p,x)$ and $f_y(p,y)$ are the probability functions of x and y, respectively, under the assumption of a free space percentage of p. Suppose a head switch costs s and there are t tracks in a cylinder, then the probability functions can be expressed as:

$$\begin{aligned} f_x(p,x) &= p(1-p)^x & (3) \\ f_y(p,y) &= f_x(1-(1-p)^{t-1}, y-s) & (4) \end{aligned}$$

In other words, f_x is the probability that there are x occupied sectors followed by a free sector in the current track, and f_y is the probability that the first $(y-s)$ rotational positions in all $(t-1)$ tracks are occupied and there is one free sector at the next rotational position in at least one of these tracks.

Figure 1 validates the model of (2) with a simulation of the HP97560 and the Seagate ST19101 disks whose relevant parameters are detailed in Table 1. The well validated HP97560 model [17, 26] is approximately eight years old. The state-of-art Seagate disk [20, 28] is more complex but its simulator is only a coarse approximation. For example,

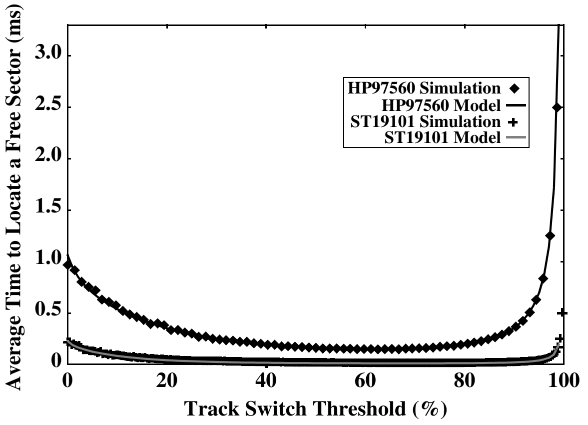

Figure 2: Average latency to locate free sectors for all writes performed to an initially empty track as a function of the track switch threshold, which is the percentage of free sectors reserved per track before a switch occurs. A high threshold corresponds to frequent switches.

the model simulates a single density zone while the actual disk has multiple zones. The simulated eager writing algorithm in Figure 1 is not restricted to the current cylinder and always seeks to the nearest sector. The figure shows that the single cylinder model is, in fact, a good approximation for an *entire* zone. This is because nearby cylinders are not much more likely than the current cylinder to have a free sector at an optimal rotational position and the head switch time is close to the single cylinder seek time.

Compared to the half-rotation delays of 7 ms for the HP and 3 ms for the Seagate that an update-in-place system may incur, Figure 1 promises significant performance improvement, especially at lower utilizations. Indeed, the figure shows that the latency has improved by nearly an order of magnitude on the newer Seagate disk compared to the HP disk. At higher disk utilizations, however, it takes longer to find a nearby free sector. One solution is to compact free space.

2.3 A Model Assuming a Compactor

Compacting free space using the disk processor can take advantage of the "free" bandwidth between the disk head and the platters during idle periods without consuming valuable I/O bus bandwidth, polluting host cache and memory, or interfering with host CPU operation. If we assume that we can compact free space during idle periods, then we need not worry about having to write to nearly full tracks. Instead, we can simply fill empty tracks to some threshold and rely on the compactor to generate more empty tracks. We will discuss the details of such compaction in Section 4.2; here we present the analysis of a system that makes such assumptions.

Suppose n is the total number of sectors in a track, m is the number of free sectors reserved per track before we switch tracks, s is the cost of a track switch, and r is the rotational delay incurred per sector, then the average latency incurred locating a free sector is:

$$\frac{s + r \cdot [(n+1)\ln\frac{n+2}{m+2} - (n-m) + \epsilon(n,m)]}{n-m} \qquad (5)$$

where the ϵ function accounts for the non-randomness of the free space distribution. Appendix A.2 provides the derivation of the model.

Figure 2 validates the model of (5). When we switch tracks too frequently, although the free sectors in any particular track between switches are plentiful, we are penalized by the high switch cost. When we switch tracks too infrequently, locating free sectors becomes harder in a crowded track and the performance is also non-optimal. In general, the model aids the judicious selection of an optimal threshold for a particular set of disk parameters. It also reassures us that we need not suffer the performance degradation seen at high utilizations. Figure 1 and 2 also show why the demands on the compactor are less than the demands on the LFS cleaner. This is because the models indicate that compacting is only necessary for high utilizations and it can move data at small granularity.

3 A Virtual Log

Eager writing allows a high degree of location independence of data. It also presents two challenges: how to locate data as its physical location can constantly change, and how to recover this location information after a failure. In this section, we explain the rationale behind the design of the virtual log and the file systems built on it.

3.1 The Indirection Map

To support location independence, we introduce an *indirection map*, which maps logical addresses to physical disk addresses, a technique similar to those found in some existing systems [5, 7, 8]. In the rest of this section, we describe how we keep the map persistent, how we perform updates, how we recover the map after a failure, and how file systems can be built using these primitives. Our goal is to avoid some of the inefficiencies and inflexibilities of the previous approaches:

- scanning large portions of the disk for recovery,

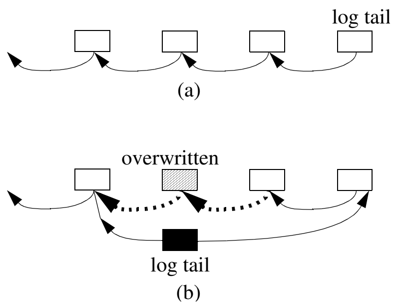

Figure 3: Maintaining the virtual log. (a) New map entry sectors are appended to a backward chain. (b) Implementing the backward chain as a tree to support map entry overwriting.

- reliance on NVRAM, which is not always available,
- excessive overhead in terms of space and extra I/O's needed to maintain the map, and
- altering the physical block format on disks to include, for example, a self-identifying header.

3.2 Maintaining and Recovering the Virtual Log

We implement the indirection map as a table. To keep the map persistent, we leverage the low latency offered by eager writing. Whenever an update takes place, we write the piece of the table that contains the new map entry to a free sector near the disk head. Suppose the file system addresses the disk at sector granularity and each map entry consumes four to eight bytes, the indirection map will consume a storage overhead between one to two percent of the disk capacity. We will discuss how to further reduce this storage overhead in the next section. If a transaction includes multiple data blocks and their map entries do not fall in the same map sector, then multiple map sectors may need to be written. Although the alternative of logging the multiple map entries using a single sector may better amortize the cost of map entry updates, it requires garbage collecting the obsolete map entries and is not used in our current design.

To recover the map after a failure, we must be able to identify the locations of the map sectors that are scattered throughout the disk due to eager writing. One way to accomplish this is to thread these sectors together to form a log. We term this a *virtual log* because its components are not necessarily physically contiguous. Because eager writing prevents us from predicting the location of the next map entry, we cannot maintain forward pointers in the map entries. Instead, we chain the map entries backward as shown in Figure 3a. Note that we can adapt this technique to a disk that supports a header per block, in which case a map entry including the backward pointer can be placed in the header.

As map entries are overwritten, the backward chain will accumulate obsolete sectors over time. We cannot simply reuse these obsolete sectors because doing so will break the backward chain. Our solution is to implement the backward linked list as a tree as shown in Figure 3b. Whenever an existing map entry is overwritten, a new log tail is introduced as the new tree root. One branch of the root points to the previous root; the other points to the map sector following the overwritten map entry. The overwritten sector can be recycled without breaking the virtual log. As Section 3.3 will show, we can keep the entire virtual log in disk memory during normal operation. Consequently, overwriting a map entry requires only one disk I/O to create the new log tail.

To recover the virtual log without scanning the disk, we must remember the location of the log tail. Modern disk drives use residual power to park their heads in a landing zone at the outer perimeter of the disks prior to spinning down the drives. It is easy to modify the firmware so that the drive records the current log tail location at a fixed location on disk before it parks the actuator[20, 31]. To be sure of the validity of the content stored at this fixed location, we can protect it with a checksum and clear it after recovery. In the extremely rare case when this power down sequence fails, we can detect the failure by computing the checksum and resort to scanning the disk for cryptographically signed map entries to retrieve the log tail.

With a stable log tail, recovering the virtual log is straightforward. We start at the log tail as the root and traverse the tree on the frontier based on age. Obsolete log entries can be recognized as such because their updated versions are younger and traversed earlier.

3.3 Implementing LFS on the Virtual Log

So far, we have described 1) a generic logging strategy that can support transactional behavior, and 2) an indirection map built on the log that can support location independence of data. One advantage of this approach is that we can implement eager writing behind a logical disk interface and deliver its performance advantage to an unmodified file system. We now describe another application of the virtual log: implementing a variant of the

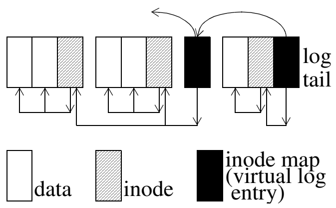

Figure 4: Implementing LFS on the virtual log. The virtual log contains only the inode map blocks.

log-structured file system (VLFS). Unlike the logical disk approach above, VLFS requires modifying the disk interface to the host file system. By seeking a tighter integration of the file system into the programmable disk, however, VLFS allows a number of optimizations impossible with an unmodified UFS. Currently, we have not directly implemented VLFS. Instead, the experiments in Section 5 are based on file systems running on the virtual log via the logical disk interface as described in the last section. We indirectly deduce the VLFS performance by evaluating these file systems.

One disadvantage of the indirection map as described in Section 3.1 is the amount of storage space and the extra I/O's needed to maintain and query the map. To solve this inefficiency, the file system can store physical addresses of the data blocks in the inodes, similar to the approach taken by LFS shown in Figure 4. As file blocks are written, the data blocks, the inode blocks that contain physical addresses of the data blocks, and inode maps that contain physical addresses of the inodes are all appended to the log. What is different in the virtual log based implementation (VLFS) is that the log need not be physically contiguous, and only the inode map blocks logically belong to the virtual log. This is essentially adding a level of indirection to the indirection map. The advantage is that the inode map, which is the sole content of the virtual log, is now compact enough to be stored in memory; it also reduces the number of I/O's needed to maintain the indirection map because VLFS simply takes advantage of the existing indirection data structures in the file system without introducing its own.

Another LFS optimization that can also be applied to VLFS is checkpointing for recovery. Periodically, we write the entire inode map to the disk contiguously. At recovery time, while LFS reads a checkpoint at a known disk location and rolls forward, VLFS traverses the virtual log backwards from the log tail towards the checkpoint.

VLFS also opens up a variety of questions including how to re-engineer the host/disk interface and how to implement the free space compactor. These issues are discussed in a technical report [35].

3.4 Comparing VLFS with LFS

VLFS and LFS share a number of common advantages. Both can benefit from an asynchronous memory buffer by preventing short-lived data from ever reaching the disk. Both can benefit from disk reorganization during idle time.

Due to eager writing, VLFS possesses a number of unique advantages. First, small synchronous writes perform well on VLFS whereas the LFS performance suffers if an application requires frequent "fsync" operations. Second, while the free space compactor is only an optimization for VLFS, the cleaner is a necessity for LFS. In cases where idle time is scarce or disk utilization is high, VLFS can avoid the bandwidth waste incurred during repeated copying of live data by the LFS cleaner [22, 29, 30]. Third, LFS needs large idle intervals to mask the cleaning overhead because it moves data at large segment granularity. The VLFS compactor, however, can take advantage of short idle intervals. Finally, reads can interfere with LFS writes by forcing the disk head away from free space and/or disturbing the track buffer (which can be sometimes used to absorb writes without accessing the disk platters). VLFS can efficiently perform intervening writes near the data being read.

VLFS and LFS also share some common disadvantages. For example, data written randomly may have poor sequential read performance. In some of these situations, reorganization techniques that can improve LFS performance [22] should be equally applicable to VLFS. In some other situations, aggressive prefetching [24] and "disk-directed I/O" [16] can also serve the virtual log well.

4 Experimental Platform

We evaluate the following four combinations of file systems and simulated disks (shown in Figure 5): a UFS on a regular disk, a UFS on a Virtual Log Disk (VLD), an LFS on a regular disk, and an LFS on a VLD. Although we do recognize that a complete evaluation of the VLFS would require a complete implementation of VLFS, in this paper, we take the first step of deducing the behavior of VLFS by examining the file systems running on the VLD.

Two host machines are used: one is a 50 Mhz SUN SPARCstation-10, which is equipped with 64 MB of memory and runs Solaris 2.6; the other is

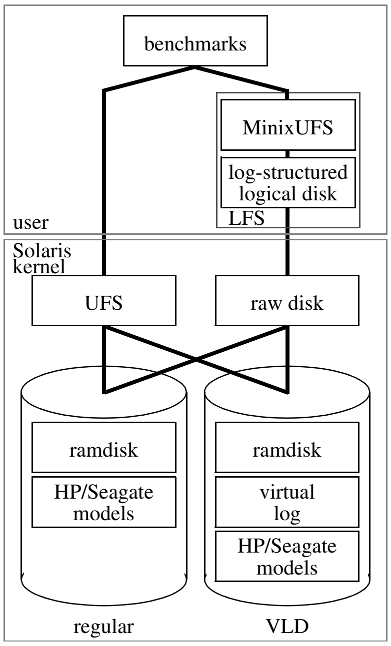

Figure 5: Architecture of the experimental platform.

a similarly configured UltraSPARC-170 workstation that runs at 167 Mhz. The SPARCstation-10 supports both 4 KB and 8 KB file blocks while the UltraSPARC-170 only supports 8 KB file blocks. Because our focus is small write performance, we run our experiments on the SPARCstation-10 unless explicitly stated to be otherwise. We perform some additional experiments on the UltraSPARC-170 only to study the impact of host processing speed. We next briefly describe each of the disk modules and file systems shown in Figure 5.

4.1 The Regular Disk

The regular disk module simulates a portion of the HP97560 disk or the Seagate ST19101 disk, whose relevant parameters are shown in Table 1. A ramdisk driver is used to store file data using 24 MB of kernel memory. The Dartmouth simulator [17] is ported into the kernel to ensure realistic timing behavior of the HP disk. We adjust the parameters of the Dartmouth model to coincide with those of the Seagate disk to simulate the faster disk. Although not as accurate, the Seagate model gives a clear indication of the impact of disk technology improvements. We have run all our experiments on both disk models. Unless stated otherwise, however, we present the results obtained on the Seagate model.

The ramdisk simulator allows two simulation modes. In one mode, the simulator sleeps the right amount of time reported by the Dartmouth model and we can conduct the evaluations by directly timing the application. In the second mode, the simulator runs at memory speed without sleeping. This mode speeds up certain phases of the experiments whose actual elapsed time is not important. The disadvantage of the ramdisk simulator is its small size due to the limited kernel memory. We only simulate 36 cylinders of the HP97560 and 11 cylinders of the Seagate.

4.2 The Virtual Log Disk

The VLD adds the virtual log to the disk simulator described above. It exports the same device driver interface so it can be used by existing file systems. The VLD maintains an in-memory indirection map while updating the on-disk representation as described in Section 3.2. To avoid trapping the disk head in regions of high utilization during eager writing, the VLD performs cylinder seeks only in one direction until it reaches the last cylinder, at which point it starts from the first cylinder again.

One challenge of implementing the VLD while preserving the existing disk interface is handling deletes, which are not visible to the device driver. This is a common problem faced by logical disks. Our solution is to monitor overwrites: when a logical address is re-used, the VLD detects that the old logical-to-physical mapping can be freed. The disadvantage of the approach is that it does not capture the freed blocks that are not yet overwritten.

Another question that arose during the implementation of the VLD is the choice of the physical disk block size. As shown in Appendix A.1, the latency is best when the physical block size matches the file system logical block size. In all our experiments, we have used a physical block size of 4 KB. The resulting internal fragmentation when writing data or metadata blocks that are smaller only biases against the performance of UFS running on the VLD. Each physical block requires a four byte map entry; so the entire map consumes 24 KB.

The third issue concerns the interaction between eager writing and the disk track buffer read-ahead algorithm. When reading, the Dartmouth simulator keeps in cache only the sectors from the beginning of the current request through the current read-ahead point and discards the data whose addresses are lower than that of the current request. This algorithm makes sense for sequential reads of data whose physical addresses increase monotonically. This is the case for traditional disk allocation strategies. For VLD, however, the combination of eager writing and the logical-to-physical address translation means that the sequentially read data may not necessarily have monotonically increasing physical addresses. As a result, the Dartmouth simulator tends to purge data prematurely from its read-ahead buffer under VLD. The solution is to aggressively prefetch the entire track as soon as the head reaches the target track and not discard data until it is delivered to the host during sequential reads. The measurements of sequential reads on the VLD in Section 5 were taken with this modification.

Lastly, the VLD module also implements a free space compactor. Although the eager writing strategy should allow the compactor to take advantage of idle intervals of arbitrary length, for simplicity, our compactor compacts free space at the granularity of tracks. During idle periods, the compactor reads the current track and uses eager writing to copy the live data to other tracks in a way similar to hole-plugging under LFS [22, 36]. Currently, we choose compaction targets randomly and plan to investigate optimal VLD compaction algorithms (e.g., ones that preserve or enhance read and write locality) in the future. Applying the lessons learned from the models in Section 2.3, the VLD fills empty tracks to a certain threshold (75% in the experiments). After exhausting empty tracks generated by the compactor, the VLD reverts to the greedy algorithm modeled in Section 2.2.

4.3 UFS

Because both the regular disk and the VLD export the standard device driver interface, we can run the Solaris UFS (subsequently also labeled as UFS) unmodified on these disks. We configure UFS with a block size of 4 KB and a fragment size of 1 KB. Like most other Unix file systems, the Solaris UFS updates metadata synchronously while the user can specify whether the data writes are synchronous. It also performs prefetching after several sequential reads are detected.

4.4 LFS

We have ported the MIT Log-Structured Logical Disk (LLD) [7], a user level implementation of LFS (subsequently also labeled as LFS). It consists of two modules: the MinixUFS and the log-structured logical disk, both running at user level. MinixUFS accesses the logical disk with a block interface while the logical disk interfaces with the raw disk using segments. The block size is 4 KB and the segment size is 0.5 MB. The implementors of LLD has disabled read-ahead in MinixUFS because blocks deemed contiguous by MinixUFS may not be so in the logical disk. A number of other issues also impact the LFS performance. First, MinixUFS employs a file buffer cache of 6.1 MB. Unless "sync" operations are issued, all writes are asynchronous. In some of the experiments in Section 5, we assume this buffer to be made of NVRAM so that the LFS configuration can have a similar reliability guarantee as that of the synchronous systems. We will examine the effectiveness of NVRAM.

Second, the logical disk's response to a "sync" operation is determined by a tunable parameter called *partial segment threshold*. If the current segment is filled above the threshold at the time of the "sync", the current segment is flushed to the disk as if it were full. If it is below the threshold, the current segment is written to the disk but the memory copy is retained to receive more writes. The partial segment threshold in the experiments is set to 75%.

Third, the original logical disk only invokes the cleaner when it runs out of empty segments. We have modified the cleaner so that it can be invoked during idle periods before it runs out of free space.

5 Experimental Results

In this section, we compare the performance of eager writing against that of update-in-place and logging with a variety of micro-benchmarks. We first run the small file and large file benchmarks that are commonly used by similar file system studies. Then we use a benchmark that best demonstrates the strength of eager writing: small random synchronous updates with no idle time, which also illustrates the effect of disk utilization. Next we examine the effect of technology trends. Last, we examine how the availability of idle time impacts the performance of eager writing and logging. Unless explicitly stated to be otherwise, the experimental platform is the SPARCstation-10 running on the simulated Seagate disk.

5.1 Small File Performance

We first examine two benchmarks similar to the ones used by both the original LFS study [25] and the Logical Disk study [7]. In the first benchmark, we create 1500 1 KB files, read them back after a cache flush, and delete them. The benchmark is run on empty disks. The results of this benchmark are shown in Figure 6. Under LFS, updates are flushed to disk only if the memory buffer is filled. Under UFS, updates are synchronous. Due to the different

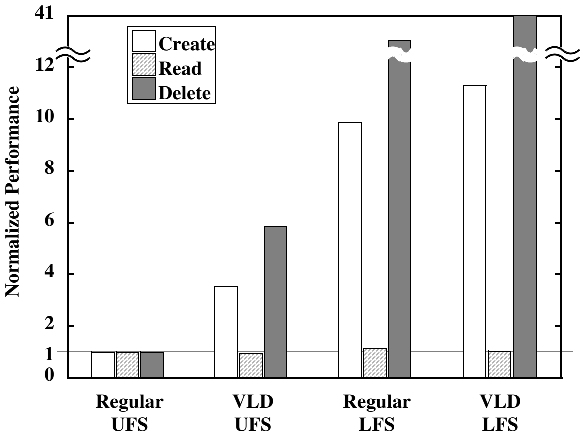

Figure 6: Small file performance. The benchmark creates, reads, and deletes 1 KB files. All performance is normalized to that of UFS running on a regular disk.

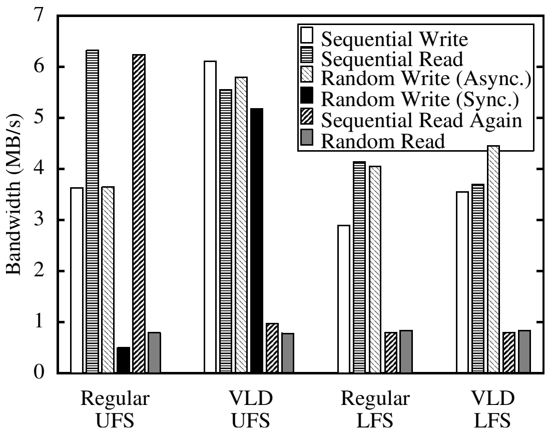

Figure 7: Large file performance. The benchmark sequentially writes a 10 MB file, reads it back sequentially, writes it again randomly (both asynchronously and synchronously for the UFS runs), reads it again sequentially, and finally reads it randomly.

reliability guarantees of UFS and LFS, this experiment is not designed to compare UFS against LFS. Instead, our interest is in examining the impact of virtual logging on both file systems.

As expected, VLD significantly speeds up UFS during the create and delete phases due to eager writing's ability to complete small writes more quickly than update-in-place. The read performance on the VLD is slightly worse than that on the regular disk because of the overhead introduced by the indirection map and the fact that disk read-ahead is not as effective. The same pattern in read performance recurs in other experiments as well.

VLD also speeds up LFS writes slightly. The reasons are the occasional miss of rotations and long-distance seeks between segment-sized writes on the regular disk, which the VLD diligently avoids.

Ignoring the cost of LFS cleaning (which is not triggered in this experiment), we speculate that the impact of VLD on a UFS that employs delayed write techniques (such as those proposed by Ganger [9]) should be between that of the unmodified UFS and that of LFS. Like LFS, however, delayed writes under UFS do not guarantee data reliability.

From this benchmark, we speculate that by integrating LFS with the virtual log, the VLFS (which we have not implemented) should approximate the performance of UFS on the VLD when we must write synchronously, while retaining the benefits of LFS when asynchronous buffering is acceptable.

5.2 Large File Performance

In the second benchmark, we write a 10 MB file sequentially, read it back sequentially, write 10 MB of data randomly to the same file, read it back sequentially again, and finally read 10 MB of random data from the file. The performance of random I/O can also be an indication of the effect of interleaving a large number of independent concurrent workloads. The benchmark is again run on empty disks. Figure 7 shows the results. The writes are asynchronous with the exception of the two random write runs on UFS that are labeled as "Sync". Neither the LFS cleaner nor the VLD compactor is run.

We first point out a few characteristics that are results of implementation artifacts. The first two phases of the LFS performance are not as good as those of UFS because the user level LFS implementation is less efficient than the native in-kernel UFS. LFS also disables prefetching, which explains its low sequential read bandwidth. Sequential reads on UFS run much faster than sequential writes on the regular disk due to aggressive prefetching both at the file system level and inside the disk. With these artifacts explained, we now examine a number of interesting performance characteristics.

First, sequential read after random write performs poorly in all LFS and VLD systems because both logging and eager writing destroy spatial locality. This is a problem that may be solved by a combination of caching, data reorganization, hints at interfaces, and prefetching as explained in Section 3.4.

Second, the LFS random write bandwidth is higher than that of sequential write. This is because during the random write phase, some of the blocks are written multiple times and fewer bytes reach the disk. This is a benefit of delayed writes.

Third, on a UFS, while it is not surprising that

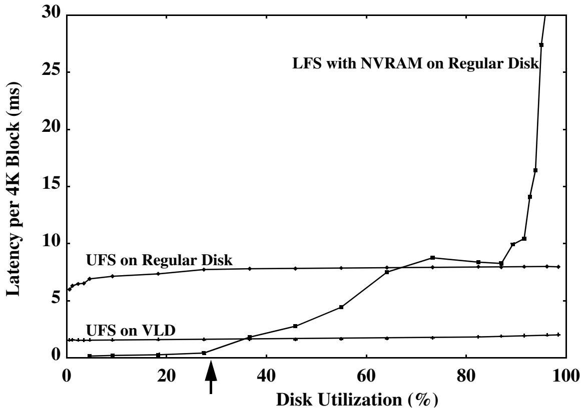

Figure 8: Performance of random small synchronous updates under various disk utilizations. The disk utilization is obtained from the Unix "df" utility and includes about 12% of reserved free space that is not usable. The arrow on the x-axis points to the size of the NVRAM used by LFS.

the synchronous random writes do well on the VLD, it is interesting to note that even sequential writes perform better on the VLD. This is because of the occasional inadvertent miss of disk rotations on the regular disk. Interestingly enough, this phenomenon does not occur on the slower HP97560 disk. This evidence supports our earlier contention that tuning the host operating system to match changing technologies is indeed a difficult task. The approach of running the file system inside the disk in general and the concept of a virtual log in particular can simplify such efforts.

Fourth, although our regular disk simulator does not implement disk queue sorting, UFS does sort the asynchronous random writes when flushing to disk. Therefore, the performance of this phase of the benchmark, which is also worse on the regular disk than on the VLD due to the reason described above, is a best case scenario of what disk queue sorting can accomplish. In general, disk queue sorting is likely to be even less effective when the disk queue length is short compared to the working set size. Similar phenomenon can happen for a write-ahead logging system whose log is small compared to the size of the database. The VLD based systems need not suffer from these limitations. In summary, the benchmark further demonstrates the power of combining lazy writing by the file system with eager writing by the disk.

5.3 Effect of Disk Utilization

There are a number of questions that are still unanswered by the first two benchmarks. First, the VLD always has plenty of free space in the previous benchmarks. How much degradation can we expect when the disk is fuller? Second, the LFS cleaner is not invoked so far. What is the impact of the cleaner under various disk utilizations? Third, we know that LFS performs poorly with frequent flushes to the disk. How much can NVRAM help? We attempt to answer these questions with the third benchmark. Again, our goal is not to compare UFS against LFS. Instead, our interest is in discovering how virtual logging can speed up synchronous writes on UFS and how NVRAM can improve LFS.

In this benchmark, we create a single file of a certain size. Then we repeatedly choose a random 4 KB block to update. There is no idle time between writes. For UFS, the "write" system call does not return until the block is written to the disk surface. For LFS, we assume that the 6.1 MB file buffer cache is made of NVRAM and we do not flush to disk until the buffer cache is full. We measure the steady state bandwidth of UFS on the regular disk, UFS on the VLD, and LFS on the regular disk as we vary the size of the file we update.

Figure 8 plots the average latency experienced per write. UFS on the regular disk suffers from excessive disk head movement due to the update-in-place policy. The latency increases slightly as the updated file grows because the disk head needs to travel a greater distance between successive writes. This increase may have been larger had we simulated the entire disk.

LFS provides excellent performance when the entire file fits in NVRAM. As soon as the file outgrows the NVRAM, writes are forced to the disk; as this happens and as disk utilization increases, the cleaner quickly dominates performance. The plateau between roughly 60% and 85% disk utilization is due to the fact that the LFS cleaner chooses less utilized segments to clean; with certain distribution patterns of free space, the number of segments to clean in order to generate the same amount of free segments may be the same as (or even larger than) that that required under a higher utilization.

With eager writing, the VLD suffers from neither the excessive disk head movements, nor the bandwidth waste during cleaning. As the disk utilization increases, the VLD latency also rises. The rise, however, is not significant compared to the various overheads in the system, which we examine next.

5.4 Effect of Technology Trends

The performance improvement provided by eager writing seen so far is not as great as the analytical models might suggest. To see why this is the case,

HP SPARC	Seagate SPARC	Seagate UltraSPARC
2.6×	5.1×	9.9×

Table 2: Performance gap between update-in-place and virtual-logging widens as disks and processors improve.

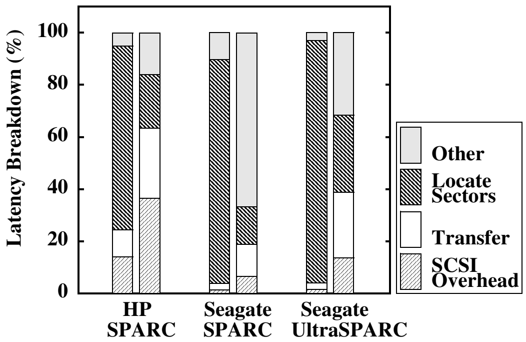

Figure 9: Breaking down the total latency into SCSI overhead, transfer time, time required to locate free sectors, and other processing time. Update-in-place performance (left bars) becomes increasingly dominated by mechanical delays while virtual logging (right bars) achieves a balance between processor and disk improvements.

we now provide a more detailed breakdown of the benchmark times reported in the last section. We also examine the impact of technology trends.

We repeat the UFS experiment of the last section on three different platforms under the same disk utilization (80%)[1]. Table 2 shows the result. The first column shows the speed-up of virtual logging compared to update-in-place on a SPARCstation-10 and HP disk combination. Next, we replace the older HP disk with the new Seagate disk. In the third run, we replace the older SPARCstation-10 with the newer UltraSPARC-170. We see that the performance gap widens to almost an order of magnitude.

Figure 9 reveals the underlying source of this performance difference by providing the detailed breakdown of the latencies. The "SCSI overhead" component is the time that the disk processor spends processing each SCSI command. The "transfer" component is the time it takes to move the bits to or from the media after the head has been positioned over the target sector. The component labeled as "locate sectors" is the amount of time the disk spends positioning the disk head. It includes seek, rotation, and head switch times. The "other" component includes the operating system processing overhead, which includes the time to run the virtual log algorithm because the disk simulator is part of the host kernel. The time consumed by simulating the disk mechanism itself is less than 5% of this component.

[1]The VLD latency in this case is measured immediately after running a compactor, as explained in Section 5.5.

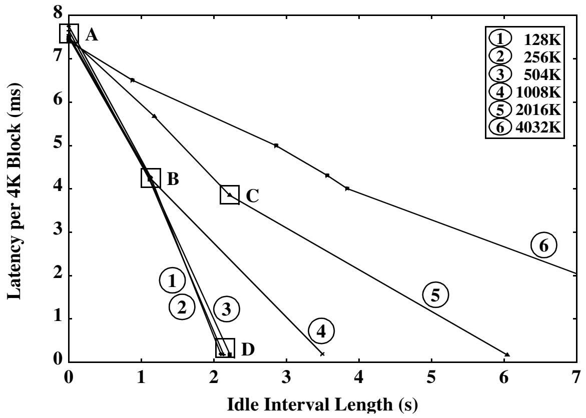

Figure 10: Performance of LFS (with NVRAM) as a function of available idle time.

We see that the mechanical delay becomes a dominant factor of update-in-place latency. We also see that eager writing has indeed succeeded in significantly reducing the disk head positioning times. The overall performance improvement on the older disk or on the older host, however, is low due to the high overheads. After we replace the older disk, the performance of virtual logging becomes host limited as the "other" component dominates. After we replace the older host, however, the latency components again become more balanced. This indicates that the virtual log is able to ride the impressive disk bandwidth growth, achieving a balance between processor and disk improvements.

5.5 Effect of Available Idle time

The benchmark in Section 5.3 assumes zero idle time. This severely stresses LFS because the cleaning time is not masked by idle periods. It also penalizes the VLD by disallowing free space compacting. In this section, we examine how LFS on a regular disk and how UFS on a VLD may benefit from idle time. We modify the benchmark of Section 5.3 to perform a burst of random updates, pause, and repeat. The disk utilization is kept at 80%.

Figure 10 shows how the LFS performance responds to the increase of idle interval length. Each curve represents a different burst size. At point A, no idle time is available. LFS fills up the NVRAM with updates and flushes the file buffer to the regular disk, invoking the cleaner when necessary.

At point B, enough idle time is available to clean one segment. If the burst size is less than or equal

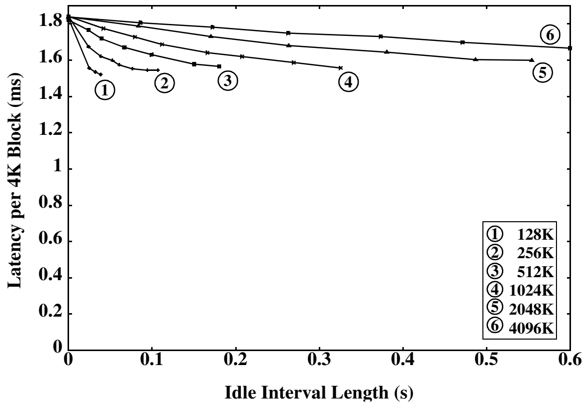

Figure 11: Performance of UFS on VLD as a function of available idle time.

to 1 MB, at the time when the NVRAM becomes full, the cleaner has already generated the maximum number of empty segments possible. Consequently, flushing the NVRAM takes a constant amount of time[2]. A similar latency is achieved at point C where two segments are cleaned per idle interval.

Point D corresponds to sufficient idle time to flush the entire burst from NVRAM to disk. The benchmark runs at memory speed because the cleaning time during flushing is entirely masked by the long idle periods. The first three curves coincide at this point because these burst sizes fit in a single segment.

We do not pretend to know the optimal LFS cleaning algorithm using NVRAM, which is outside the scope of this paper. Nevertheless, two facts are apparent. First, because the cleaner moves segment-sized data, LFS can only benefit from relatively long idle intervals. Second, unless there is sufficient idle time to mask the flushing of the burst buffered in the NVRAM, LFS performance remains poor.

Figure 11 shows the result of repeating the same benchmark on UFS running on a VLD. Unlike the LFS cleaner, the VLD compactor has no restriction in terms of the granularity of the data it moves. For convenience, our implementation of the compactor moves data at track granularity, which is still much smaller than the LFS segment granularity. The performance on the VLD improves along a continuum of relatively small idle intervals. The VLD performance is also more predictable, whereas LFS performance experiences large variances depending on whether the NVRAM is full and whether cleaning is necessary. The experiments of this section run on the SPARCstation-10. As we have seen in the last section, the more powerful UltraSPARC-170 can easily cut the latency in Figure 11 in half.

[2]In this case, because the NVRAM size is larger than the available free space, the cleaner still needs to run during flushing to reclaim the free space created during flushing.

The disadvantage of UFS on the VLD compared to LFS with NVRAM is the limiting performance with infinite amount of idle time. Because each write reaches the disk surface, the VLD experiences the overheads detailed in Section 5.4. The overheads also render the impact of the compactor less important. Fortunately, as explained in that section, as technology improves, we expect the gap of the limiting performance to narrow.

Furthermore, eager writing does not dictate the use of a UFS, nor does it preclude the use of NVRAM. We speculate that a VLFS with NVRAM can enjoy 1) the low latency of NVRAM and its filtering effect on short-lived data, and 2) the effective use of bandwidth using eager writing when idle intervals are short or disk utilization is high.

6 Related Work

The work presented in this paper builds upon a number of existing techniques including reducing latency by writing data near the disk head, a transactional log, file systems that support data location independence, and log-structured file systems. The goal of this study is not to demonstrate the effectiveness of any of these individual ideas. Rather, the goals are 1) provide a theoretical foundation of eager writing with the analytical models, 2) show that the integration of these ideas at the disk level can provide a number of unique benefits to both UFS and LFS, 3) demonstrate the benefits of eager writing without the semantic compromise of delayed writes or extra hardware support such as NVRAM, and 4) conduct a series of systematic experiments to quantify the differences of the alternatives. We are not aware of existing studies that aim for these goals.

Simply writing data near the disk head is not a new idea. Many efforts have focused on improving the performance of the write-ahead log. This is motivated by the observation that appending to the log may incur extra rotational delay even when no seek is required. The IBM IMS Write Ahead Data Set (WADS) system [10] addresses this issue for drums (fixed-head disks) by keeping some tracks completely empty. Once each track is filled with a single block, it is not re-used until the data is copied out of the track into its normal location.

Likewise, Hagmann places the write-ahead log in its own logging disk[13], where each log append can fill any open block in the cylinder until its utilization reaches a threshold. Eager writing in our system, while retaining good logging performance, assumes no dedicated logging disk and does not require copying of data from the log into its permanent location. The virtual log *is* the file system.

A number of disk systems have also explored the idea of lowering latency of small writes by writing near the disk head location. Menon proposes to use this technique to speed up parity updates in disk arrays [23]. Under *parity logging*, an update is performed to a rotationally optimal position in a cylinder. It relies on NVRAM to keep the indirection map persistent.

Mime [5], the extension of Loge [8], also writes near disk head and is the closest in spirit to our system. There are a number of differences between Mime and the virtual log. First, Mime relies on self-identifying disk blocks. Second, Mime scans free segments to recover its indirection map. As disk capacity increases, this scanning may become a time consuming process. Third, the virtual log also incorporates a free space compactor.

The Network Appliance file system, WAFL [14, 15], checkpoints the disk to a consistent state periodically, uses NVRAM for fast writes between checkpoints, and can write data and metadata anywhere on the disk. An exception of the write-anywhere policy is the root inodes, which are written for each checkpoint and must be at fixed locations. Unlike Mime, WAFL supports fast recovery by rolling forward from a checkpoint using the log in the NVRAM. One goal of the virtual log is to support fast transactions and recovery without NVRAM, which has capacity, reliability, and cost limitations. Another difference is that the WAFL write allocation decisions are made at the RAID controller level, so the opportunity to optimize for rotational delay is limited.

Our idea of fast atomic writes using the virtual log originated as a generalization of the AutoRAID technique of hole-plugging [36] to improve LFS performance at high disk utilizations without AutoRAID hardware support for self-describing disk sectors. In hole-plugging, partially empty segments are freed by writing their live blocks into the holes found in other segments. This outperforms traditional cleaning at high disk utilizations by avoiding reading and writing a large number of nearly full segments [22]. AutoRAID requires an initial log-structured write of a physically contiguous segment, after which it is free to copy the live data in a segment into any empty block on the disk. In contrast, the virtual log eliminates the initial segment write and can efficiently schedule the individual writes.

7 Conclusion

In this paper, we have developed a number of analytical models that show the theoretical performance potential of eager writing, the technique of performing small atomic writes near the disk head position. We have presented the virtual log design which supports fast synchronous writes without the semantic compromise of delayed writes or extra hardware support such as NVRAM. This design requires careful log management for fast recovery. By conducting a systematic comparison of this approach against traditional update-in-place and logging approaches, we have demonstrated the benefits of applying the virtual log approach to both UFS and LFS. The availability of fast synchronous writes can also simplify the design of file systems and sophisticated applications. As the current technology trends continue, we expect that the performance advantage of this approach will become increasingly important.

Acknowledgements

We would like to thank Arvind Krishnamurthy for many interesting discussions on the proofs of several analytical models, Marvin Solomn for discovering the simplest proof of the single track model, Daniel Stodolsky and Chris Malakapalli for helping us understand the working of Quantum and Seagate disks, Jeanna Neefe Matthews for asking the question of how hole-plugging could be efficiently implemented without hardware support for self-describing disk sectors, Geoff Voelker and John Wilkes for comments on early drafts and presentations, and Margo Seltzer for a large number of excellent suggestions during the shepherding process.

References

[1] Adams, L., and Ou, M. Processor Integration in a Disk Controller. *IEEE Micro 17*, 4 (July 1997).

[2] Atkinson, M., Chisholm, K., Cockshott, P., and Marshall, R. Algorithms for a Persistent Heap. *Software - Practice and Experience 13*, 3 (March 1983), 259–271.

[3] Baker, M., Asami, S., Deprit, E., Ousterhout, J., and Seltzer, M. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the*

Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V) (Sept. 1992), pp. 10–22.

[4] Birrell, A., Hisgen, A., Jerian, C., Mann, T., and Swart, G. The Echo Distributed File System. Technical Report 111, Digital Equipment Corp. Systems Research Center, Sept. 1993.

[5] Chao, C., English, R., Jacobson, D., Stepanov, A., and Wilkes, J. Mime: a High Performance Parallel Storage Device with Strong Recovery Guarantees. Tech. Rep. HPL-CSP-92-9 rev 1, Hewlett-Packard Company, Palo Alto, CA, March 1992.

[6] Chutani, S., Anderson, O., Kazar, M., Leverett, B., Mason, W., and Siedbotham, R. The Episode File System. In *Proc. of the 1992 Winter USENIX* (January 1992), pp. 43–60.

[7] de Jonge, W., Kaashoek, M. F., and Hsieh, W. C. The Logical Disk: A New Approach to Improving File Systems. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 15–28.

[8] English, R. M., and Stepanov, A. A. Loge: a Self-Organizing Disk Controller. In *Proc. of the 1992 Winter USENIX* (January 1992).

[9] Ganger, G. R., and Patt, Y. N. Metadata Update Performance in File Systems. In *Proc. of the First Symposium on Operating Systems Design and Implementation* (November 1994), pp. 49–60.

[10] Gawlick, D., Gray, J., Limura, W., and Obermarck, R. Method and Apparatus for Logging Journal Data Using a Log Write Ahead Data Set. U.S. Patent 4507751 issued to IBM, March 1985.

[11] Grochowski, E. G., and Hoyt, R. F. Future Trends in Hard Disk Drives. *IEEE Transactions on Magnetics 32*, 3 (May 1996).

[12] Hagmann, R. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proc. of the 11th ACM Symposium on Operating Systems Principles* (October 1987), pp. 155–162.

[13] Hagmann, R. Low Latency Logging. Tech. Rep. CSL-91-1, Xerox Corporation, Palo Alto, CA, February 1991.

[14] Hitz, D., Lau, J., and Malcolm, M. File System Design for an NFS File Server Appliance. In *Proc. of the 1994 Winter USENIX* (January 1994).

[15] Hitz, D., Lau, J., and Malcolm, M. File System Design for an NFS File Server Appliance. Tech. Rep. 3002, Network Appliance, March 1995.

[16] Kotz, D. Disk-directed I/O for MIMD Multiprocessors. In *Proc. of the First Symposium on Operating Systems Design and Implementation* (November 1994), pp. 61–74.

[17] Kotz, D., Toh, S., and Radhakrishnan, S. A Detailed Simulation Model of the HP 97560 Disk Drive. Tech. Rep. PCS-TR91-220, Dept. of Computer Science, Dartmouth College, July 1994.

[18] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. The ObjectStore Database System. *Communications of the ACM 34*, 10 (October 1991), 50–63.

[19] Lowell, D. E., and Chen, P. M. Free Transactions with Rio Vista. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (October 1997), pp. 92–101.

[20] Malakapalli, C. Personal Communication, Seagate Technology, Inc., July 1998.

[21] Mashey, J. R. Big Data and the Next Wave of InfraStress. Computer Science Division Seminar, University of California, Berkeley, October 1997.

[22] Matthews, J. N., Roselli, D. S., Costello, A. M., Wang, R. Y., and Anderson, T. E. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proc. of the 16th ACM Symposium on Operating Systems Principles* (October 1997), pp. 238–251.

[23] Menon, J., Roche, J., and Kasson, J. Floating parity and data disk arrays. *Journal of Parallel and Distributed Computing 17*, 1 and 2 (January/February 1993), 129–139.

[24] Patterson, R. H., Gibson, G. A., Ginting, E., Stodolsky, D., and Zelenka, J. Informed Prefetching and Caching. In *Proceedings of the ACM Fifteenth Symposium on Operating Systems Principles* (December 1995).

[25] Rosenblum, M., and Ousterhout, J. The Design and Implementation of a Log-Structured File System. In *Proc. of the 13th Symposium on Operating Systems Principles* (Oct. 1991), pp. 1–15.

[26] Ruemmler, C., and Wilkes, J. An Introduction to Disk Drive Modeling. *IEEE Computer 27*, 3 (March 1994), 17–28.

[27] Satyanarayanan, M., Mashburn, H. H., Kumar, P., Steere, D. C., and Kistler, J. J. Lightweight Recoverable Virtual Memory. In *Proc. of the 14th ACM Symposium on Operating Systems Principles* (December 1993), pp. 146–160.

[28] Seagate Technology, Inc. Cheetah Specifications. http://204.160.183.162/disc/cheetah/cheetah.shtml, 1998.

[29] Seltzer, M., Bostic, K., McKusick, M., and Staelin, C. An Implementation of a Log-Structured File System for UNIX. In *Proc. of the 1993 Winter USENIX* (Jan. 1993), pp. 307–326.

[30] Seltzer, M., Smith, K., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V. File System Logging Versus Clustering: A Performance Comparison. In *Proc. of the 1995 Winter USENIX* (Jan. 1995).

[31] Stodolsky, D. Personal Communication, Quantum Corp., July 1998.

[32] Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., and Peck, G. Scalability in the XFS File System. In *Proc. of the 1996 Winter USENIX* (January 1996), pp. 1–14.

[33] Transaction Processing Performance Council. *TPC Benchmark B Standard Specification.* Waterside Associates, Fremont, CA, Aug. 1990.

[34] Transaction Processing Performance Council. *TPC Benchmark C Standard Specification.* Waterside Associates, Fremont, CA, August 1996.

[35] Wang, R. Y., Anderson, T. E., and Patterson, D. A. Virtual Log Based File Systems for a Programmable Disk. Tech. Rep. UCB/CSD 98/1031, University of California at Berkeley, December 1998.

[36] Wilkes, J., Golding, R., Staelin, C., and Sullivan, T. The HP AutoRAID Hierarchical Storage System. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (December 1995), pp. 96–108.

A.1 Proof and Extension of the Single Track Model

Suppose a disk track contains n sectors, its free space percentage is p, and the free space is randomly distributed. Section 2.1 gives the average number of sectors the disk head must skip before arriving at a free sector:

$$\frac{(1-p)n}{1+pn} \tag{6}$$

Proof. Suppose k is the number of free sectors in the track and $E(n,k)$ is the expected number of used sectors we encounter before reaching the first free sector. With a probability of k/n, the first sector is free and the number of sectors the disk head must skip is 0. Otherwise, with a probability of $(n-k)/n$, the first sector is used and we must continue searching in the remaining $(n-1)$ sectors, of which k are free. Therefore, the expected delay in this case is $[1+E(n-1,k)]$. This yields the recurrence:

$$E(n,k) = \frac{n-k}{n}[1+E(n-1,k)] \tag{7}$$

By induction on n, it is easy to prove that the following is the unique solution to (7):

$$E(n,k) = \frac{n-k}{1+k} \tag{8}$$

(6) follows if we substitute k in (8) with pn. □

Formula (6) models the latency to locate a single sector (or a block). To extend it to situations where we need to find multiple disk sectors (or blocks) to satisfy a single file system block allocation, consider the two following examples. First, let us suppose that the host file system writes 4 KB blocks. If the disk allocates and frees 512 B sectors, it may need to locate eight separate free sectors to complete eager writing one logical block. If the disk uses a physical block size of 4 KB instead, although it takes longer to locate the first free 4 KB-aligned sector, it is guaranteed to have eight consecutive free sectors afterwards. In general, suppose the file system logical block size is B and the disk physical block size is b ($b \leq B$), then the average amount of time (expressed in the numbers of sectors skipped) needed to locate all the free sectors for a logical block is:

$$\frac{(1-p)n}{b+pn} \cdot B \tag{9}$$

Formula (9) indicates that the latency is lowest when the physical block size matches the logical block size.

A.2 Derivation of the Model Assuming a Compactor

We derive the model of Section 2.3. If writes arrive with no delay, then it would be trivial to fill the track. If writes arrive randomly and we assume the free space distribution is random at the time of the arrivals, then writes between successive track switches follow the model of (6). Therefore, substituting pn in (6) with i, the number of free sectors, the total number of sectors to skip between track switches can be expressed as:

$$\sum_{i=m+1}^{n} \frac{n-i}{1+i} \tag{10}$$

where n is the total number of sectors in a track and m is the number of free sectors reserved per track before switching tracks. Suppose each track switch costs s and the rotation delay of one sector is r; because we pay one track switch cost per $(n-m)$ writes, the average latency per sector of this strategy can be expressed as:

$$\frac{s + r \cdot \sum_{i=m+1}^{n} \frac{n-i}{1+i}}{n-m} \tag{11}$$

So far, we have assumed that the free space distribution is always random. This is not the case with the approach of filling the track up to a threshold because, for example, the first free sector immediately following a number of used sectors is more likely to be selected for eager writing than one following a number of free sectors. In general, this non-randomness increases latency. Although the precise modeling of the non-randomness is quite complex, through approximations and empirical trials, we have found that adding the following ϵ function to (10) works well for a wide range of disk parameters in practice:

$$\epsilon(n,m) = \frac{(n-m-0.5)^{p+2}}{(8-\frac{n}{96}) \cdot (p+2) \cdot n^{p}} \tag{12}$$

where $p = 1 + n/36$. By approximating the summation in (10) with an integral and adding the correction factor of (12) to account for the non-randomness, we arrive at the final latency model:

$$\frac{s + r \cdot [(n+1)\ln\frac{n+2}{m+2} - (n-m) + \epsilon(n,m)]}{n-m} \tag{13}$$