INFORMATION AND COLLABORATION IN THE STORAGE STACK

by

Timothy Edward Denehy

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Science)

at the

UNIVERSITY OF WISCONSIN–MADISON

2006

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Though there have been substantial innovations in both file systems and storage systems over the past twenty-five years, the interface with which they communicate has remained simple and abstract. The storage stack that exists today was not developed in a coherent manner; rather, it evolved over time due to the flexible nature of this abstraction. The result is an *information gap*: the file system no longer understands the nature of its underlying storage, and the storage system cannot comprehend the semantics of the blocks it stores. In the end, this *obscuring interface* has lead to the development of independent, locally optimized, complex layers whose interactions may lead to poor performance and limited functionality.

Therefore, we believe it is time to re-examine the structure of the storage stack and the division of labor between the file system and storage system layers. Specifically, we advocate designs that enable vertical coordination and even *collaboration* between layers in order to achieve the goals of the entire storage stack. Furthermore, we believe the key to achieving these goals lies in information, and therefore depends on the development of *informing interfaces* that facilitate such vertical designs.

In this dissertation, we take three steps in this direction. First, we develop a system that automatically discovers the properties of a RAID storage system using only the logical block abstraction, transforming the obscuring interface into a basic informing interface. Second, we examine the applicability of such storage level information by designing an improved informing interface and a file system that explicitly manages the multiple disks of a storage array. Third, we develop a vertical design for collaboration between journaling file systems and RAID systems that improves the reliability and availability of the overall storage system.

Chapter 1

Introduction

A chasm exists in the world of file storage and management. Though a hierarchical file system of directories and byte-accessible files has been the norm for almost 30 years [52], the internals of both file systems and underlying storage systems have evolved substantially.

In file systems, many approaches have been developed to improve performance, including read-optimized inode and file placement [40], logging of writes [55], improved meta-data update methods [67], more scalable internal data structures [75], and off-line reorganization strategies [39]. However, almost all such techniques have been developed under the assumption that the file system will be run upon a single, traditional disk.

Storage systems have also received much attention from the research community. One of the most notable innovations is the disk array, or RAID [47]. These storage systems contain multiple disks and internally manage both parallelism and redundancy to optimize for performance, capacity, or even both [88]. Today, RAID disk arrays have become the dominant form of storage for high-end applications. These modern storage systems are increasingly complex. For example, an enterprise storage array can contain tens of processors and hundreds of disks [21], and a given array can be configured in many different ways.

While the changes in both file systems and storage systems have been substantial, they have also been separate, and the result is an *information gap*: the file system does not understand the true nature of the storage system it runs upon, and the storage system cannot comprehend the semantic relations between the blocks it stores. In addition, the layers are unaware of the particular responsibilities, functionalities, and optimizations being performed by the other. This gap arose from a historical source: the boundary between software and hardware and the requisite interface for communicating across it.

The predominant storage interface dates to the introduction of the Small Computer System Interface (SCSI) standard in 1986. SCSI defined a simple logical block address (LBA) interface for disks instead of the device level interfaces that were common at the time. This new interface was independent of the geometry of the device, allowing companies to develop interoperable systems and peripherals. This abstract view of storage enabled much of the innovation above and below the SCSI interface. New file systems and storage devices could be developed independently and yet used together because they shared a common logical interface.

In fact, the development of RAID systems in particular was enabled by the flexibility of this abstract interface. Regardless of their internal complexity, RAID arrays expose the same simple interface as a single disk: a linear array of blocks accessible via read and write operations. Storage vendors took advantage of the freedom to innovate behind this interface, and thus developed high-performance, high-capacity systems that appeared as a single, large, and fast disk to the file system. No software modifications were required of the host operating system, and file systems continued to operate correctly, in spite of the fact that they were often optimized for a single-disk system. In this case, ignorance was bliss; the arrangement was simple and worked well.

We term this arrangement of a file system layer on top of a storage layer a *storage stack,* akin to networking protocol stacks that are prominent in communication networks [17]. There are some similarities between the two: layering is known to simplify system design, though potentially at the cost of performance [83]. However, a crucial difference exists: the layers that comprise network protocol stacks are derived by design, with the architects carefully deciding where each specific element should be placed.

The storage stack, on the other hand, has not been developed in a single, coherent manner. Quite to the contrary, the storage stack that exists today has *evolved* over time based on the existence of a flexible interface and the objectives of the storage industry. As more and more complexity is introduced at each layer, the potential for detrimental and difficult-to-predict interactions increases. These may manifest not only in poor performance, but also in duplication and implementation complexity, competition between layers, and limitations on functionality.

For example, performance may suffer if the model that the file system has of the storage layer is not accurate; thus, layout optimizations that work well on a single, traditional disk may not be appropriate when the logical-block to physical-block mapping is unknown. In fact, the layout decisions made by the file system level may compete with (and be overruled by) those made in an advanced storage system [36, 88].

Feature duplication is also a potential pitfall. For example, a log-structured file system [55] or a journaling file system [82] could be layered on top of a disk array that also performs logging [73, 88], degrading performance, duplicating work, and increasing system complexity unnecessarily. Similarly, a large block cache at the storage level may be rendered ineffective if it largely duplicates the contents of the file system buffer cache [89].

Finally, functionality may be limited, as certain pieces of information only live at one layer of the system. For example, the storage system does not know what blocks constitute a file and thus it cannot perform per-file operations. Similarly, the storage system does not know that a particular block no longer contains live data after a file deletion, and thus it cannot optimize operations that may ignore dead blocks (*e.g.* RAID reconstruction).

1.1 Approach

In the end, the abstract, block-based, *obscuring interface* to storage has lead to the development of independently designed and optimized layers that are oblivious to their role in the overall storage stack. Thus, we believe it is time to re-examine the division of labor between the file system and storage system layers, in an attempt to understand the best way to structure the storage stack. Specifically, we advocate designs that enable vertical coordination and even *collaboration* between layers in order to achieve the goals of the entire storage stack. Furthermore, we believe the key to achieving these goals lies in information, and therefore depends on the development of *informing interfaces* that facilitate such vertical designs.

1.2 Deconstructing Storage Arrays

We begin by proposing a basic informing interface that simply exports the configuration parameters of a RAID, including the number of disks, chunk size, level of redundancy, and layout scheme. This interface gives the file system or other client enough information to determine the mapping of blocks from the logical address space to individual disks. It also conveys the amount of reliability that is implemented by the array and the particular scheme (*e.g.* RAID-1 or RAID-5).

This approach is simple in that it merely reports information that already exists in the array, rather than requiring implementation of new mechanisms, analogous to the Infokernel [3] approach for operating systems interfaces. Therefore, the onus to make use of this information is placed on the file system. For example, the file system may alter its policies to make use of the individual disks, or it might modify its access pattern to avoid deficiencies in the RAID scheme.

The most efficient implementation of this informing interface would be for storage arrays to support a query that returns their configuration parameters. However, it may be difficult to convince storage vendors to add this interface in a standard way, as even small extensions require industry deliberation and consensus. Thus, we take a pragmatic approach and treat the RAID as a gray box [2], inferring its characteristics and configuration using only its existing logical block interface.

To do so, we introduce Shear, a user-level software system that characterizes RAID storage arrays. Shear employs a set of controlled workloads combined with statistical techniques to automatically determine the RAID configuration parameters that constitute our basic informing interface (the number of disks, chunk size, level of redundancy, and layout scheme). We illustrate the correctness of Shear by running it upon numerous simulated configurations, and then verify its real-world applicability by running Shear on both software-based and hardware-based RAID systems.

We also demonstrate the utility of Shear and the basic informing interface through three case studies. First, we show how Shear can be used in a storage management environment to verify RAID construction and detect failures. Second, we demonstrate how the interface can be used to extract detailed characteristics about the individual disks within an array. Third, we show how an operating system can use the informing interface to automatically tune its storage subsystems to specific RAID configurations.

1.3 Bridging the Information Gap

Although our basic informing interface has shown to be useful, it provides details at a rather low level. File systems that want to take advantage of the array configuration must be imbued with particular knowledge of each possible RAID scheme and its unique performance and reliability characteristics. Given the number of RAID variants that exist today, and the potential growth of new schemes in the future, designing a file system that can account for such a large population may prove difficult.

To overcome this limitation, we introduce a second informing interface, Exposed RAID, that encapsulates array information in abstractions that are meaningful to file system objectives. Specifically, the E×RAID address space is divided into a set of regions, each of which is mapped to a single disk or a set of disks. Hence, these regions represent the performance and failure boundaries within the disk array. In addition to this static information, E×RAID provides dynamic information about the performance and reliability of each region that may be exploited by the file system to manage its use of the storage.

We make use of the E×RAID informing interface to evaluate a new division of labor between the storage system and the file system. In particular, we design an Informed Log-Structured File System (I·LFS) that explicitly manages and takes advantage of the performance and failure boundaries present in a multiple disk storage system. Experiments reveal that our prototype implementation yields benefits in the management, flexibility, reliability, and performance of the storage system, with only a small increase in file system complexity. For example, I·LFS can incorporate new disks into the system on-the-fly, dynamically balance workloads across the disks of the system, allow for user control of file replication, and delay replication of files for increased performance. Much of this functionality would be difficult to implement with the existing relationship between file systems and storage systems predicated on the traditional logical block interface.

1.4 Collaborating Layers

Finally, we look beyond information-only interfaces to new mechanisms that allow storage stack layers to communicate more effectively. Specifically, we develop a collaborative approach using a journaling file system to address the problem of slow, scan-based, software RAID resynchronization that restores consistency after a system crash. We analyze Linux ext3 and introduce a new mode of operation, declared mode, that guarantees to provide a record of all outstanding writes in case of a crash. To utilize this information, we augment the software RAID with an informing interface (verify read) that instructs the RAID layer to inspect and repair the redundant information for a block. The combination of these features allows us to provide fast, journal-guided resynchronization. We evaluate the effect of journal-guided resynchronization and find that it provides improved software RAID reliability and availability after a crash, while suffering little performance loss during normal operation.

1.5 Contributions

- The development of a set of algorithms and analyses that automatically determine the configuration properties of a RAID array using only the logical block interface, and the embodiment of these techniques in a software system (Shear) that transforms the existing interface into a basic informing interface.

- The design and implementation of an informing interface (E$\times$RAID) that provides meaningful abstractions to a new file system (I·LFS) that explicitly manages the individual components of a disk array to improve the performance and functionality of the overall system.

- The design and implementation of a collaborative approach between a journaling file system and a software RAID layer to provide fast, journal-guided resynchronization after a crash that improves both the reliability and the availability of the storage system.

1.6 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we introduce Shear and its algorithms and apply it to the management and performance tuning of disk arrays. Chapter 3 presents E$\times$RAID and I·LFS and their use of disk information to improve the performance and functionality of the storage stack. In Chapter 4, we develop journal-guided resynchronization and evaluate its ability to improve the reliability and availability of the storage system. We present related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Deconstructing Storage Arrays with Shear

Modern storage systems are complex. For example, a high-end storage array can contain tens of processors and hundreds of disks [21] and a given array can be configured in many different ways, most commonly using RAID-0, RAID-1, or RAID-5. However, regardless of their internal complexity, RAID arrays expose a simple interface consisting of a linear array of blocks. All of the internal complexity is hidden; a large array exports exactly the same interface as a single disk.

This encapsulation has many advantages, the most important of which is *transparent* operation of unmodified file systems on top of any storage device. But this transparency also has a cost: users and applications cannot easily obtain more information about the complexities of the storage system because of the *obscuring interface*. For example, most storage systems do not reveal how data blocks are mapped to each of the underlying disks, and it is well known that RAID configuration has a large impact on performance and reliability [13, 47, 61, 88]. Furthermore, despite the fact that configuring a modern array is difficult and error-prone, administrators are given little help in verifying the correctness of their setup.

To overcome this information gap, we propose a basic informing interface that simply exports the configuration parameters of a RAID, including the number of disks, chunk size, level of redundancy, and layout scheme. This interface gives the file system or other client enough information to determine the mapping of blocks from the logical address space to individual disks. It also conveys the amount of reliability that is implemented by the array and the particular scheme (*e.g.* RAID-1 or RAID-5).

This approach is simple in that it merely reports information that already exists in the array, rather than requiring implementation of new mechanisms, analogous to the Infokernel [3] approach for operating systems interfaces. Therefore, the onus to make use of this information is placed on the file system. For example, the file system may alter its policies to make use of the individual disks, or it might modify its access pattern to avoid deficiencies in the RAID scheme.

The most efficient implementation of this informing interface would be for storage arrays to support a query that returns their configuration parameters. However, it may be difficult to convince storage vendors to add this interface in a standard way, as even small extensions require industry deliberation and consensus. Thus, we take a pragmatic approach and treat the RAID as a gray box [2], inferring its characteristics and configuration using only its existing logical block interface.

In this chapter, we describe Shear, a user-level software system that automatically identifies the important RAID configuration parameters of our basic informing interface. Using this system to characterize a RAID allows developers of higher-level software, including file systems and database management systems, to tailor their implementations to the specifics of the array upon which they run. Further, administrators can use Shear to understand details of their arrays, verifying that they have configured the RAID as expected or even observing that a disk failure has occurred.

As is common in microbenchmarking, the general approach used by Shear is to generate controlled I/O request patterns to the disk and to measure the time the requests take to complete. Indeed, others have applied generally similar techniques to single-disk storage systems [62, 76, 90]. By carefully constructing these I/O patterns, Shear can derive a broad range of RAID array characteristics, including details about block layout strategy and redundancy scheme.

In building Shear, we applied a number of general techniques that were critical to its successful realization. Most important was the application of *randomness*; by generating random I/O requests to disk, Shear is better able to control its experimental environment, thus avoiding a multitude of optimizations that are common in storage systems. Also crucial to Shear is the inclusion of a variety of *statistical clustering techniques*; through these techniques, Shear can automatically come to the necessary conclusions and thus avoid the need for human interpretation.

We demonstrate the effectiveness of Shear by running it upon both simulated and real RAID configurations. With simulation, we demonstrate the breadth of Shear, by running it upon a variety of configurations and verifying its correct behavior. We then show how Shear can be used to discover interesting properties of real systems. By running Shear upon the Linux software RAID driver, we uncover a poor method of parity updates in its RAID-5 mode. By running Shear upon an Adaptec 2200S RAID controller, we find that the card uses the unusual left-asymmetric parity scheme [34].

Finally, we demonstrate the utility of the Shear system and the basic informing interface through three case studies. In the first, we show how administrators can use Shear to verify the correctness of their configuration and to determine whether a disk failure has occurred within the RAID array. Second, we demonstrate how Shear and the basic interface enable existing tools [62, 76, 90] to extract detailed information about individual disks in an array. Third, we show how a file system can use knowledge of the underlying RAID to improve performance. Specifically, we show that a modified Linux ext2 file system that performs *stripe-aware writes* improves sequential I/O performance on a hardware RAID by over a factor of two.

The rest of this chapter is organized as follows. In Section 2.1 we describe Shear, illustrating its output on a variety of simulated configurations and redundancy schemes. Then, in Section 2.2, we show the results of running Shear on software and hardware RAID systems, and in Section 2.3, we show how Shear can be used to improve storage administration and file system performance through three case studies. We conclude in Section 2.5.

2.1 Shear

We now describe Shear, our software for identifying the characteristics of a storage system containing multiple disks. We begin by describing our assumptions about the underlying storage system. We then present details about the RAID simulator that we use to both verify Shear and to give intuition about its behavior. Finally, we describe the algorithms that compose Shear.

2.1.1 Assumptions

In this chapter, we focus on characterizing block-based storage systems that are composed of multiple disks. Specifically, given certain assumptions, Shear is able to determine the mapping of logical block numbers to individual disks as well as the disks for mirrored copies and parity blocks. Our model of the storage system captures the common RAID levels 0, 1, 4, and 5, and variants such as P+Q [13] and chained declustering [28].

We assume a storage system with the following properties. Data is allocated to disks at the block level, where a *block* is the minimal unit of data that the file system reads or writes from the storage system. A *chunk* is a set of blocks that is allocated contiguously within a disk; we assume a constant chunk size. A *stripe* is a set of chunks across each of D data disks.

Shear assumes that the mapping of logical blocks to individual disks follows some repeatable, but unknown, pattern. The *pattern* is the minimum sequence of data blocks such that block offset i within the pattern is always located on disk j; likewise, the pattern's associated mirror and parity blocks, i_m and i_p, are always on disks k_m and k_p, respectively. Note that in some configurations, the pattern size is identical to the stripe size (*e.g.*, RAID-0 and RAID-5 left-symmetric), whereas in others the pattern size is larger (*e.g.*, RAID-5 left-asymmetric). Based on this assumption, Shear cannot detect more complex schemes, such as AutoRAID [88], that migrate logical blocks among different physical locations and redundancy levels.

Figure 2.1 illustrates a number of the layout configurations that we analyze in this chapter. Each configuration contains four disks and uses a chunk size of four blocks, but we vary the layout algorithm and the level of redundancy.

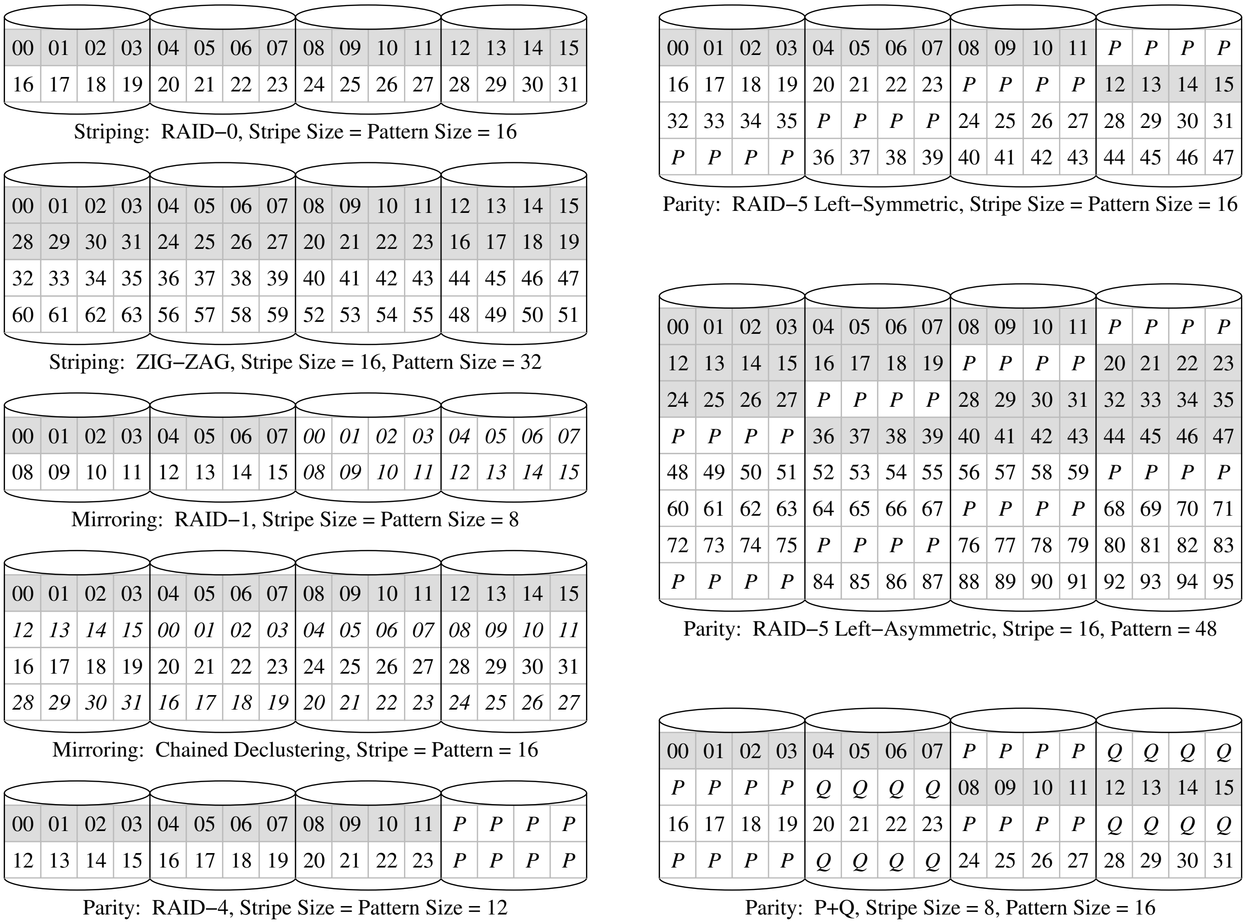


Figure 2.1 **Examples and Terminology.** This picture displays a number of four disk arrays using several of the layout patterns discussed in this chapter. The numbers represent blocks, P and Q indicate parity blocks, and redundant data is denoted with italics. In each case, the chunk size is four blocks and the stripe size and pattern size in blocks are listed. Each array depicts at least two full patterns for the given layout scheme, the first of which is shaded in gray.

RAID systems typically contain significant amounts of memory for caching. Shear currently does not attempt to identify the amount of storage memory or the policy used for replacement; however, techniques developed elsewhere may be applicable [11, 62, 84, 90]. Due to its use of random accesses and steady-state behavior, Shear operates correctly in the presence of a cache, as long as the cache is small relative to the storage array. With this assumption, Shear is able to initiate new read requests that are not cached and perform writes that overwhelm the capacity of the cache.

Our framework makes a few additional assumptions. First, we assume that all of the disks are relatively homogeneous in both performance and capacity. However, the use of random accesses again makes Shear more robust to heterogeneity, as described in more detail below. Second, we assume that Shear is able to access the raw device; that is, it can access blocks directly from the storage system, bypassing the file system and any associated buffer cache. Finally, we assume that there is little traffic from other processes in the system; however, we have found that Shear is robust to small perturbations.

2.1.2 Techniques

The basic idea of Shear is that by accessing sets of disk blocks and timing those accesses, one is able to detect which blocks are located on the same disks and thus infer basic properties of block layout. Intuitively, sets of reads that are "slow" are assumed to be located on the same disk; sets of reads that are "fast" are assumed to be located on different disks. Beyond this basic approach, Shear employs a number of techniques that are key to its operation.

Randomness: The key insight employed within Shear is to use random accesses to the storage device. Random accesses are important for a number of reasons. First, random accesses increase the likelihood that each request will actually be sent to a disk (*i.e.*, is not cached or prefetched by the RAID). Second, the performance of random access is dominated by the number of disk heads that are servicing the requests; thus Shear is able to more easily identify the number of disks involved. Third, random accesses are less likely to saturate interconnects and hide performance differences. Finally, random accesses tend to homogenize the performance of slightly heterogeneous disks: historical data indicates that disk bandwidth improves by nearly 40% per year, whereas seek time and rotational latency improve by less than 10% per year [25]; as a result, disks from different generations are more similar in terms of random performance than sequential performance. Note that, in the actual implementation, a pseudo-random number generator is used to produce the set of disk accesses.

Steady-state: Shear measures the steady-state performance of the storage system by issuing a large number of random reads or writes (*e.g.*, approximately 500 outstanding requests). Examining steady-state performance ensures that the storage system is not able to prefetch or cache all of the requests. This is especially important for write operations that could be temporarily buffered in a write-back RAID cache.

Statistical inferences: Shear automatically identifies the parameters of the storage system with statistical techniques. Although Shear provides graphical presentations of the results for verification, a human user is not required to interpret the results. This automatic identification is performed by clustering the observed access times with K-means and X-means [48]; this clustering allows Shear to determine which access times are similar and thus which blocks are correlated.

Safe operations: All of the operations that Shear performs on the storage system are safe; most of the accesses are read operations and those that are writes are performed by first reading the existing data into memory and then writing out the same data (assuming exclusive access to the array). As a result, Shear can be run on storage systems containing live data and this allows Shear to inspect RAIDs that appear to have disk failures or other performance anomalies over time.

2.1.3 Simulation Framework

To demonstrate the correct operation of Shear, we have developed a storage system simulator. We are able to simulate storage arrays with a variety of striping, mirroring, and parity configurations; for example, we simulate RAID-0, RAID-1, RAID-4, RAID-5 with left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric layouts [34], P+Q redundancy [13], and chained declustering [28]. We can configure the number of disks and the chunk size per disk. The storage array can also include a cache.

The disks within the storage array are configured to perform similarly to an IBM 9LZX disk. The simulation of each disk within the storage array is fairly detailed, accurately modeling seek time, rotation latency, track and cylinder skewing, and a simple segmented cache. We have configured our disk simulator through a combination of three methods [62]: issuing SCSI commands and measuring the elapsed time, by directly querying the disk, and by using the values provided by the manufacturer. Specifically, we simulate a rotation time of 6 ms, head switch time of 0.8 ms, a cylinder switch time of 1.8 ms, a track skew of 36 sectors, a cylinder skew of 84 sectors, 272 sectors per track, and 10 disk heads. The seek time curve is modeled using the two-function equation proposed by Ruemmler and Wilkes [57]; for short seek distances (less than 400 cylinders) the seek time is proportional to the square root of the cylinder distance (with endpoints at 0.8 and 6.0 ms), and for longer distances the seek time is proportional to the cylinder distance (with endpoints of 6.0 and 8.0 ms).

2.1.4 Algorithm

Shear has four steps; in each step, a different parameter of the storage system is identified. First, Shear determines the pattern size. Second, Shear identifies the boundaries between disks as well as the chunk size. Third, Shear extracts more detailed information about the actual layout of blocks to disks. Finally, Shear identifies the level of redundancy.

Although Shear behaves correctly with striping, mirroring, and parity, the examples in this section begin by assuming a storage system without redundancy. We show how Shear operates with redundancy with additional simulations in Section 2.1.5. We now describe the four algorithmic steps in more detail.

2.1.4.1 Pattern Size

In the first step, Shear identifies the pattern size. This *pattern size*, P, is defined as the minimum distance such that, for all B, blocks B and $B+P$ are located on the same disk. Shear operates

```
for p in 1 to maximum pattern size {
    choose a random offset o_r between 0 and p − 1
    for i in 1 to N {
        choose a random segment s based on pattern size p
        create a request for offset o_r in s
    }
    issue all requests in parallel and time their completion
}
```

Figure 2.2 **Pattern Size Detection Algorithm.**

by testing for an assumed pattern size, varying the assumed size p from a single block up to a pre-defined maximum (a slight but unimplemented refinement would simply continue until the desired output results). For each p, Shear divides the storage device into a series of non-overlapping, consecutive segments of size p. Then Shear selects a random segment offset, o_r, along with N random segments, and issues parallel reads to the same offset o_r within each segment. This workload of random requests is repeated R times and the completion times are averaged. Increasing N has the effect of concurrently examining more segments on the disk; increasing R conducts more trials with different random offsets. Pseudo-code for the algorithm is shown in Figure 2.2.

The intuition behind this algorithm is as follows. By definition, if p does not match the actual pattern size, P, then the requests will be sent to different disks; if p is equal to P, then all of the requests will be sent to the same disk. When requests are serviced in parallel by different disks, the response time of the storage system is expected to be less than that when all requests are serviced by the same disk.

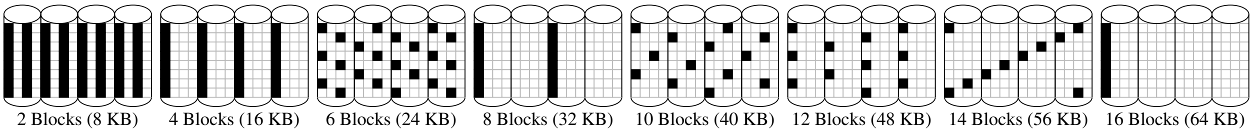


Figure 2.3 **Pattern Size Detection: Sample Execution.** Given 4 disks and a chunk size of 4 blocks, the shaded blocks are read as Shear increments the assumed pattern size. For compactness, the figure starts with an assumed pattern size of 2 blocks and increases each time by 2 blocks. The figure highlights all blocks at the given stride; in reality, only N random blocks are read.

To illustrate this behavior, we consider a four disk RAID-0 array with a block size of 4 KB and a chunk size of 4 blocks (16 KB); thus, the actual pattern size is 16 blocks (64 KB). Figure 2.3 shows the location of the reads as the assumed pattern size is increased for a sample execution. The top graph of Figure 2.4 shows the corresponding timings when this workload is run on the simulator.

The sample execution shows that when the assumed pattern is 2, 4, or 6 blocks, the requests are sent to all disks; as a result, the timings with a stride of 8, 16, and 24 KB are at a minimum. The sample execution next shows that when the assumed pattern is 8 blocks, the requests are sent to only two disks; as a result, the timing at 32 KB is slightly higher. Finally, when the assumed pattern size is 16 blocks, all requests are sent to the same disk and a 64 KB stride results in the highest time.

To detect pattern size automatically, Shear clusters the observed completion times using a variant of the X-means cluster algorithm [48]; this clustering algorithm does not require that the number of clusters be known *a priori*. Shear then selects that cluster with the greatest mean completion time. The correct pattern size, P, is calculated as the greatest common divisor of the pattern size assumptions in this cluster.

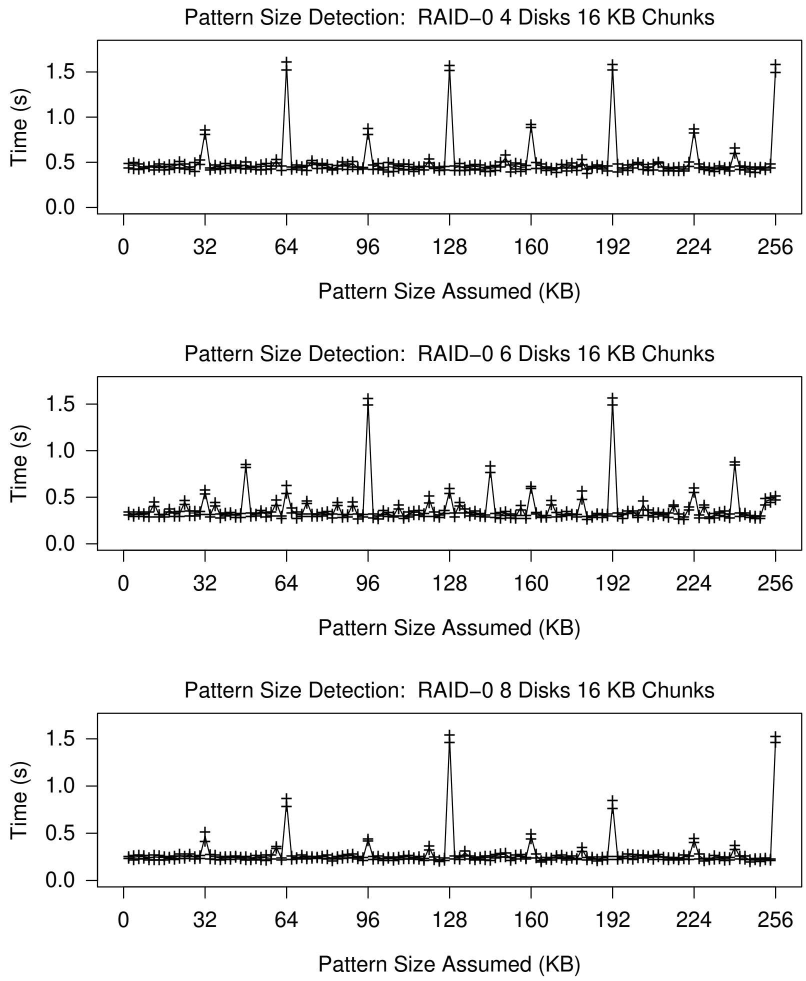


Figure 2.4 **Pattern Size Detection: Simulations.** The graphs show the results of running the pattern size detection algorithm on RAID-0 with 16 KB chunks and 4, 6, and 8 disks.

To demonstrate that Shear is able to detect different pattern sizes, we configure the simulator with six and eight disks in the remaining two graphs of Figure 2.4. As desired, blocks with a stride of 96 KB (*i.e.*, 6 disks × 16 KB) and 128 KB (*i.e.*, 8 disks × 16 KB) are located on the same disk and mark the length of the pattern.

2.1.4.2 Boundaries and Chunk Size

In the second step, Shear identifies the data boundaries between disks and the chunk size. A data boundary occurs between blocks a and b when block a is allocated to one disk and block b to another. The chunk size is defined as the amount of data that is allocated contiguously within a single disk.

Shear operates by assuming that a data boundary occurs at an offset, c, within the pattern. Shear then varies c from 0 to the pattern size determined in the previous step. For each c, Shear selects N patterns at random and creates a read request for offset c within the pattern; Shear then selects another N random patterns and creates a read request at offset $(c - 1)\ mod\ P$. All $2N$ requests for a given c are issued in parallel and the completion times are recorded. This workload is repeated for R trials and the times are averaged. Pseudo-code for the algorithm is shown in Figure 2.5.

The intuition is that if c does not correspond to a disk boundary, then all of the requests are sent to the same disk and the workload completes slowly; when c does correspond to a disk boundary, then the requests are split between two disks and complete quickly (due to parallelism).

To illustrate, we consider the same four disk RAID-0 array as above. Figure 2.6 shows a portion of a sample execution of the chunk size detection algorithm and the top graph of Figure 2.7 shows

```
for c in 0 to P − 1 {
   for i in 1 to N {
      choose a random pattern p based on pattern size P
      create a request for block c in p
   }
   for i in 1 to N {
      choose a random pattern p based on pattern size P
      create a request for block (c − 1) mod P in p
   }
   issue all requests in parallel and time their completion
}
```

Figure 2.5 **Chunk Size Detection Algorithm.**

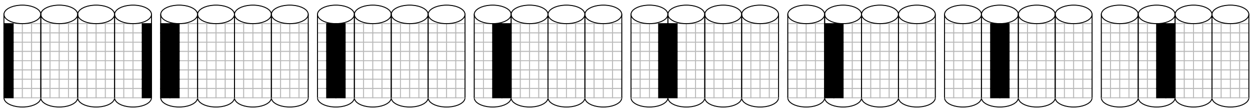


Figure 2.6 **Chunk Size Detection: Sample Execution.** Given 4 disks and 4 block chunks, the shaded blocks are read as Shear increments the offset within the pattern. Although requests are shown accessing every pattern, only N are selected at random.

the timings. The sample execution shows that when c is equal to 0 and 4, the requests are sent to different disks; for all other values of c, the requests are sent to the same disk. The timing data validates this result in that requests with an offset of 0 KB and 16 KB are faster than the others.

Shear automatically determines the chunk size C by dividing the observed completion times into two clusters using the K-Means algorithm and selecting the cluster with the smallest mean completion time. The data points in this cluster correspond to the disk boundaries; the RAID chunk size is calculated as the difference between these boundaries.

To show that Shear can detect different chunk sizes, we consider a few striping variants. We begin with RAID-0 and a constant pattern size (*i.e.*, 64 KB); we examine both 8 disks with 8 KB chunks and 16 disks with 4 KB chunks in the next two graphs in Figure 2.7. As desired, the accesses are slow at 8 KB and 4 KB intervals, respectively. To further stress boundary detection, we consider ZIG-ZAG striping in which alternating stripes are allocated in the reverse direction; this scheme is shown in Figure 2.1. The last graph shows that the first and last chunks in each stripe appear twice as large, as expected.

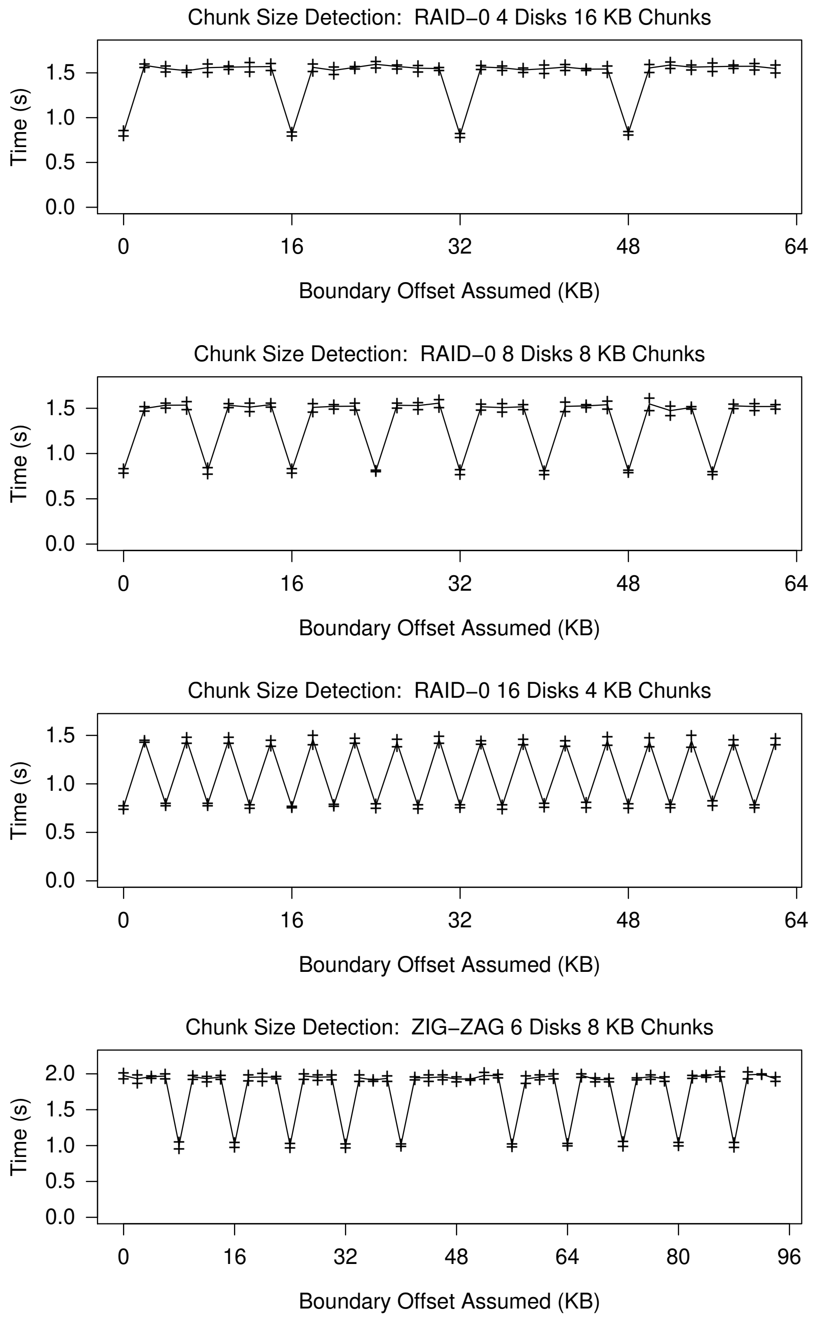


Figure 2.7 **Chunk Size Detection: Simulations.** The first three graphs use RAID-0 configurations: 4 disks with 16 KB chunks, 8 disks with 8 KB chunks, and 16 disks with 4 KB chunks. The last graph uses the ZIG-ZAG striping configuration in which alternating stripes are allocated in the reverse direction; this has 6 disks and 8 KB chunks.

```
divide the pattern into n = P/C chunks
for c1 in 0 to n − 1 {
    for c2 in c1 to n − 1 {
        for i in 1 to N {
            choose a random pattern p based on pattern size P
            create a request for the first block of chunk c1 in p
        }
        for i in 1 to N {
            choose a random pattern p based on pattern size P
            create a request for the first block of chunk c2 in p
        }
        issue all requests in parallel and time their completion
    }
}
```

Figure 2.8 **Layout Detection Algorithm.**

2.1.4.3 Layout

The previous two steps allow Shear to determine the pattern size and the chunk size. In the third step, Shear infers which chunks within the repeating pattern fall onto the same disk.

To determine which chunks are allocated to the same disk, Shear examines in turn each pair of chunks, c_1 and c_2, in a pattern. First, Shear randomly selects N patterns and creates read requests for chunk c_1 within each pattern; then Shear selects another N patterns and creates read requests for c_2 within each pattern. All of the requests for a given pair are issued in parallel and the completion times are recorded. This workload is repeated over R trials and the results are averaged. Shear then examines the next pair. Pseudo-code for the algorithm is shown in Figure 2.8.

Figure 2.9 shows that these results can be visualized in an interesting way. For these experiments, we configure our simulator to model both RAID-0 and ZIG-ZAG with 6 disks and 8 KB

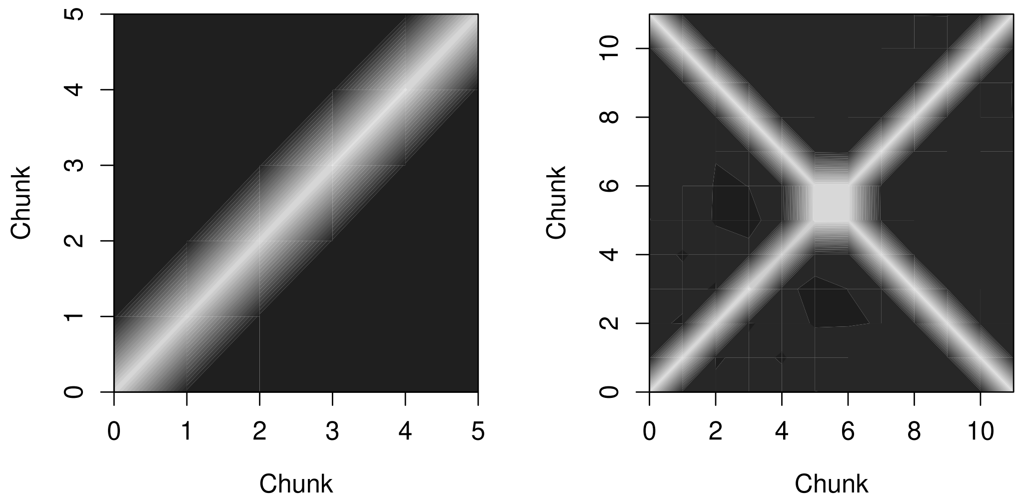


Figure 2.9 **Read Layout Detection: Simulations.** The first graph uses RAID-0; the second graph uses ZIG-ZAG. Both configurations use 6 disks and 8 KB chunks. The points in the graph correspond to pairs of chunks within a pattern that are accessed simultaneously. Lighter points indicate the workload finished more slowly and therefore those chunks reside on the same disk.

chunks. Each point in the graph corresponds to a (c_1, c_2) pair; light points indicate slow access times and thus fall on the same disk. The diagonal line in each graph corresponds to pairs where $c_1 = c_2$ and thus always fall on the same disk. In RAID-0, no chunks within a pattern are allocated to the same disk; thus, no pairs are shown in conflict. However, in ZIG-ZAG, the second half of each pattern conflicts with the blocks in the first half, shown as the second (upper-left to lower-right) diagonal line.

To automatically determine which chunks are on the same disk, Shear divides the completion times into two clusters using K-means and selects the cluster with the largest mean completion time. Shear infers that the chunk pairs from this cluster are on the same physical disk. By dividing the chunks into associative sets, Shear can infer the number of primary data disks in the system.

The above algorithm elicits read dependencies between pairs of chunks. Running the same algorithm with writes instead of reads allows Shear to identify write dependencies, which may occur due to rotating mirrors as in chained declustering or a shared parity block in RAID-4 or RAID-5. For example, consider the RAID-5 left-asymmetric array in Figure 2.1. Writing to blocks 0 and 16 at the same time will result in a short response time because the operations are spread across all four disks. Writing to blocks 0 and 52, however, will result in a longer response time because they share a parity disk. Similarly, writing to blocks 0 and 20 will take longer because the parity block for block 0 resides on the same disk as block 20.

The write layout results can reinforce conclusions from the read layout results, and they will be used to distinguish between RAID-4, RAID-5, and P+Q, as well as between RAID-1 and chained declustering. We discuss write layouts further and provide example results in Section 2.1.5.

2.1.4.4 Redundancy

In the fourth step, Shear identifies how redundancy is managed within the array. Generally, the ratio between random read bandwidth and random write bandwidth is determined by how the disk array manages redundancy.

Therefore, to detect how redundancy is managed, Shear compares the bandwidth for random reads and writes. Shear creates N block-sized random reads, issues them in parallel, and times their completion. Shear then times N random writes issued in parallel; these writes can be performed safely if needed, by first reading that data from the storage system and then writing out the same values (with extra intervening traffic to flush any caches). The ratio between the read and write bandwidth is then compared to our expectations to determine the amount and type of redundancy.

For storage arrays with no redundancy (*e.g.*, RAID-0), the read and write bandwidths are expected to be approximately equal. For storage systems with a single mirror (*e.g.*, RAID-1), the read bandwidth is expected to be twice that of the write bandwidth, since reads can be balanced across mirrored disks but writes must propagate to two disks. More generally, the ratio of read bandwidth to write bandwidth exposes the number of mirrors. For systems with RAID-5 parity, write bandwidth is roughly one fourth of read bandwidth, since a small write requires reading the existing disk contents and the associated parity, and then writing the new values back to disk. In RAID-4 arrays, however, the bandwidth ratio varies with the number of disks because the single parity disk is a bottleneck. This makes RAID-4 more difficult to identify, and we discuss this further in Section 2.2.

One problem that arises in our redundancy detection algorithm is that instead of solely using reads, Shear also uses writes. Using writes in conjunction with reads is essential to Shear as it allows us to observe the difference between the case when a block is being read and the case when a block (and any parity or mirrors) is being committed to disk.

Unfortunately, depending on the specifics of the storage system under test, writes may be buffered for some time before being written to stable storage. Some systems do this at the risk of data loss (*e.g.*, a desktop drive that has immediate reporting enabled), whereas higher-end arrays may have some amount of non-volatile RAM that can be used to safely delay writes that have been acknowledged. In either case, Shear needs to avoid the effects of buffering and move to the steady-state domain of inducing disk I/O when writes are issued.

The manner in which Shear achieves this is through a simple, adaptive technique. The basic idea is that during the redundancy detection algorithm, Shear monitors write bandwidth during the write phase. If write performance is more than twice as fast as the previously observed read performance, Shear concludes that some or all of the writes were buffered and not written to disk, so another round of writes is initiated. Eventually, the writes will flood the write cache and induce the storage system into the desired steady-state behavior of writing most of the data to disk; Shear detects this transition by observing that writes are no longer much faster than reads (indeed, they are often slower). We explore this issue more thoroughly via experimentation in Section 2.2.

2.1.4.5 Identifying Known Layouts

Finally, Shear uses the pattern size, chunk size, read layout, write layout, and redundancy information in an attempt to match its observations to one of its known schemes (e.g. RAID-5 left-asymmetric). If a match is found, Shear first re-evaluates the number of disks in the system. For instance, the number of disks will be doubled for RAID-1 and incremented for RAID-4. Shear completes by reporting the total number of disks in the array, the chunk size, and the layout observed.

If a match is not found, Shear reports the discovered chunk size and number of disks, but reports that the specific algorithm is unknown. By assuming that chunks are allocated sequentially to disks, Shear can produce a suspected layout based on its observations.

2.1.5 Redundancy Simulations

In this section, we describe how Shear handles storage systems with redundancy. We begin by showing results for systems with parity, specifically RAID-4, RAID-5, and P+Q. We then consider mirroring variants: RAID-1 and chained declustering. In all simulations, we consider a storage array with six disks and an 8 KB chunk size. For the purpose of comparison, we present a base case of RAID-0 in Figure 2.10.

2.1.5.1 Parity

Shear handles storage systems that use parity blocks as a form of redundancy. To demonstrate this, we consider four variants of RAID-5, RAID-4, and P+Q redundancy [13].

RAID-5: RAID-5 calculates a parity block for each stripe of data, and the location of the parity block is rotated between disks. RAID-5 can have a number of different layouts of data and parity blocks, such as left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric [34]. Left-symmetric is known to deliver the best bandwidth [34], and is the only layout in which the pattern size is equal to the stripe size (*i.e.*, the same as for RAID-0); in the other RAID-5 layouts, the pattern size is $D - 1$ times the stripe size.

The pattern size results for the four RAID-5 systems are shown in Figure 2.11. The first graph shows that the pattern size for left-symmetric is 48 KB, which is identical to that of RAID-0; the other three graphs show that left-asymmetric, right-symmetric, and right-asymmetric have pattern sizes of 240 KB (*i.e.*, 30 chunks), as expected. Note that despite the apparent noise in the graphs,

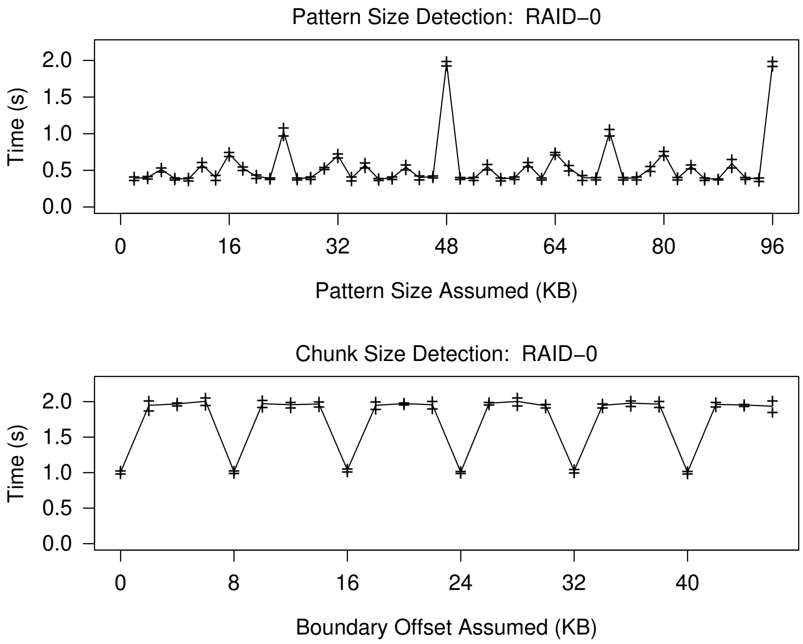


Figure 2.10 **Pattern Size and Chunk Size Detection: RAID-0.** We simulate RAID-0 with 6 disks and 8 KB chunks. The first graph confirms that the pattern size is 48 KB; the second graph confirms that the chunk size is 8 KB.

the X-means clustering algorithm is able to correctly identify the pattern sizes. The chunk size algorithm does not behave differently for RAID-5 versus RAID-0; therefore we omit those results.

Figure 2.12 shows the read layout and write layout graphs for RAID-5. Note that each of the four RAID-5 variants leads to a very distinct visual image. As before, light points correspond to dependent chunk pairs that are slow; points that are dark correspond to independent chunk pairs that offer fast concurrent access. A read dependence occurs when the two chunks are located on the same disk. Write dependencies occur when the two chunks reside on the same disk, share a parity disk, or cause interference with a parity disk. These instances result in an overburdened disk and a longer response time.

Each graph depicts a pattern-sized grid that accounts for all possible pairs of chunks. For example, the RAID-5 left-asymmetric read layout graph is a 30 chunk by 30 chunk grid. The points that pair chunk 0 with chunks 5, 10, 15, 20, and 25 are all light in color because those chunks are located on the same disk. With this knowledge, Shear is able to identify if the storage system is using one of these standard RAID-5 variants and it can calculate the number of disks.

RAID-4: RAID-4 also calculates a single parity block for each stripe of data, but all of the parity blocks reside on a single disk. The pattern size, chunk size, read layout, and write layout results for RAID-4 are shown in Figure 2.13. The pattern size is 40 KB because the parity disk is invisible to the read-based workload. The read layout graph resembles the RAID-0 result because the pattern size is equal to the stripe size, and therefore each disk occurs only once in the pattern.

On the other hand, the write layout graph for RAID-4 is quite unique. Because the parity disk is a bottleneck for writes, all pairs of chunks are limited by a single disk and therefore exhibit similar

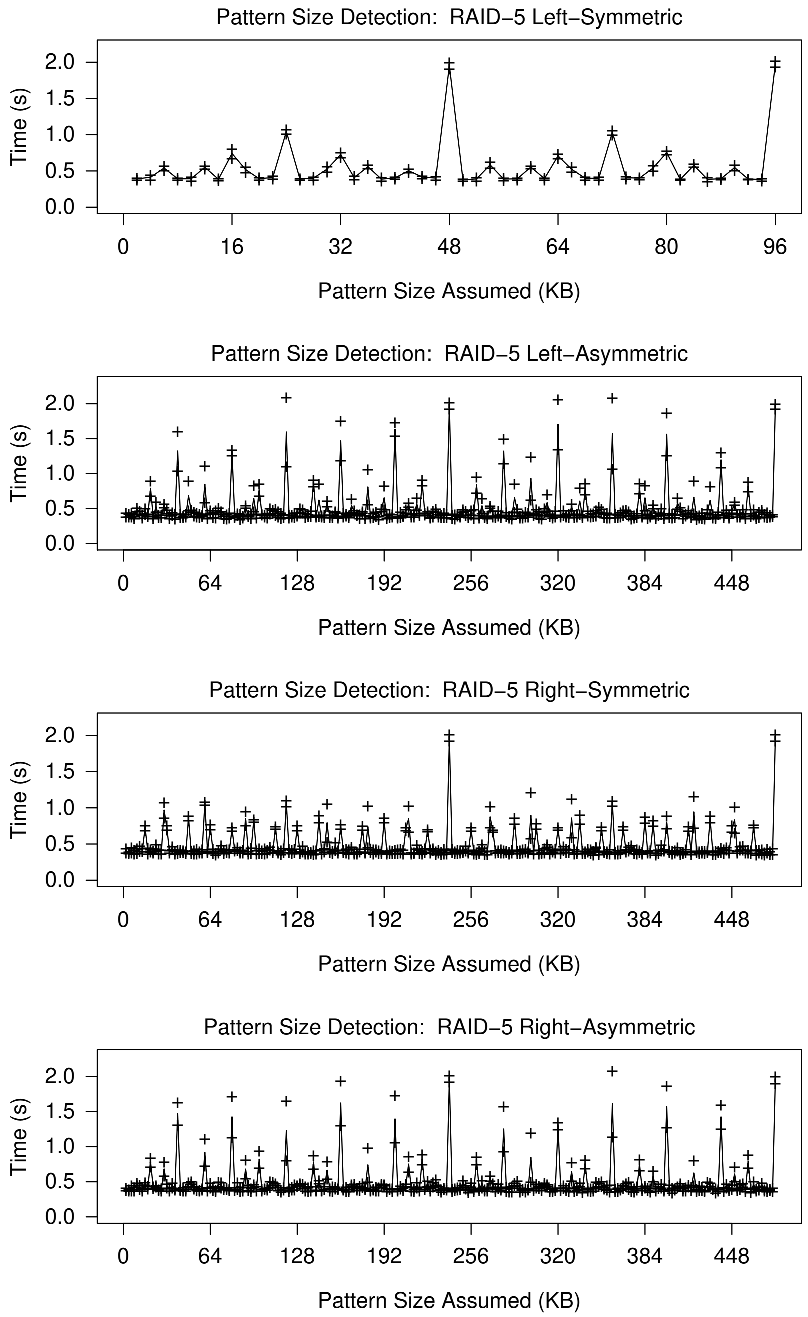


Figure 2.11 **Pattern Size Detection: RAID-5.** We simulate RAID-5 with left-symmetric, left--asymmetric, right-symmetric, and right-asymmetric layouts. Each configuration uses 6 disks and a chunk size of 8 KB. The pattern size is 48 KB for RAID-5 left-symmetric and 240 KB for the rest.

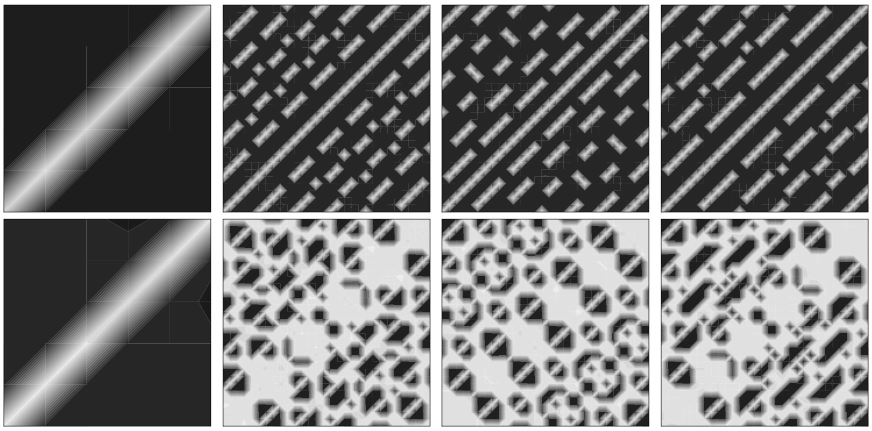


Figure 2.12 **Read and Write Layout Detection: RAID-5.** We simulate (from left to right) RAID-5 left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric, with 6 disks. The first row displays the read layouts and the second row shows the write layout graphs.

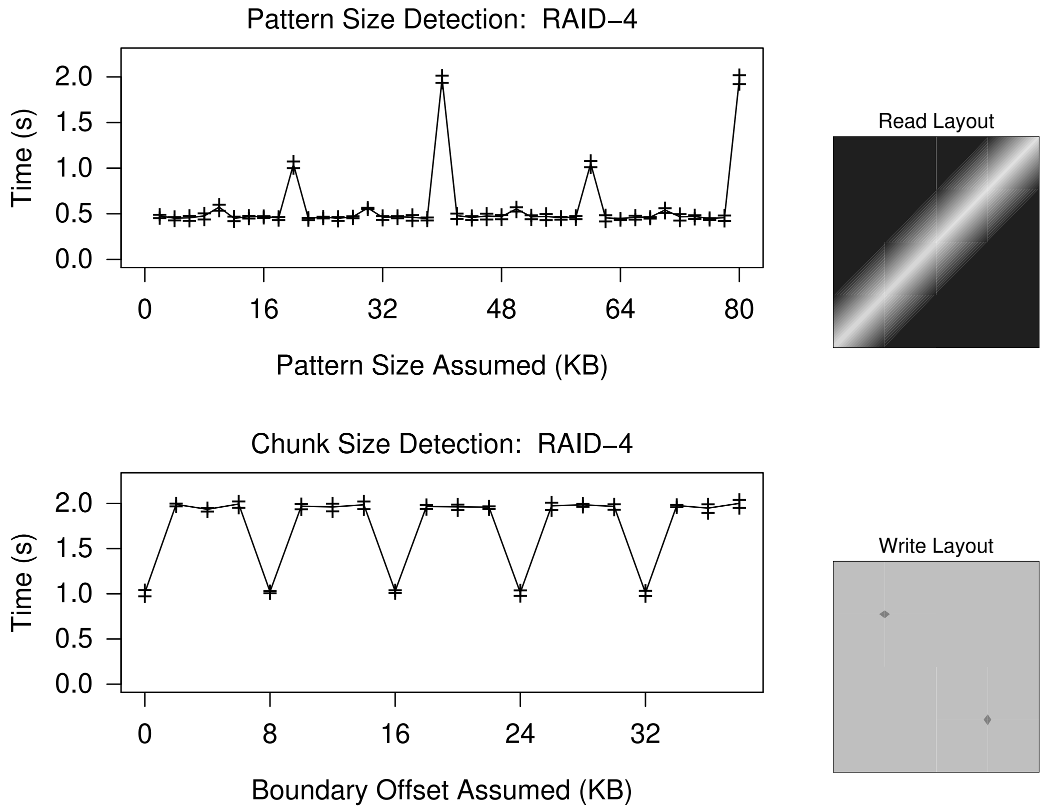


Figure 2.13 **Pattern Size, Chunk Size, and Layout Detection: RAID-4.** We simulate RAID-4 with 6 disks and 8 KB chunks. The first graph confirms that the pattern size of 40 KB is detected; the second graph shows the chunk size of 8 KB is detected. The read layout graph on the right resembles that for RAID-0, but the write layout graph uniquely distinguishes RAID-4 from other parity-based schemes.

completion times. This bottleneck produces a relatively flat RAID-4 write layout graph, allowing us to distinguish RAID-4 from other parity schemes.

P+Q: To demonstrate that Shear handles other parity schemes, we show the results of detecting pattern size and chunk size for P+Q redundancy (RAID-6). In this parity scheme, each stripe has two parity blocks calculated with Reed-Solomon codes; otherwise, the layout looks like left-symmetric RAID-5. In Figure 2.14, the first graph shows that the pattern size of 48 KB is detected; the second graph shows an 8 KB chunk size.

Figure 2.14 also shows the read layout and write layout graphs for P+Q. The read layout graph resembles that for RAID-0. The write layout graph, however, exhibits three distinct performance regions. The slowest time occurs when all requests are sent to the same chunk (and disk) in the repeating pattern. The fastest time occurs when the requests and parity updates are spread evenly across four disks, for instance when pairing chunks 0 and 1. A middle performance region occurs when parity blocks for one chunk conflict with data blocks for the other chunk. For example, when testing chunks 0 and 2, about half of the parity updates for chunk 2 will fall on the disk containing chunk 0. Again, this unique write layout allows us to distinguish P+Q from the other parity-based schemes.

2.1.5.2 Mirroring

Using the same algorithms, Shear can also handle storage systems that contain mirrors. However, the impact of mirrors is much greater than that of parity blocks, since read traffic can be directed to mirrors. A key assumption we make is that reads are balanced across mirrors; if reads

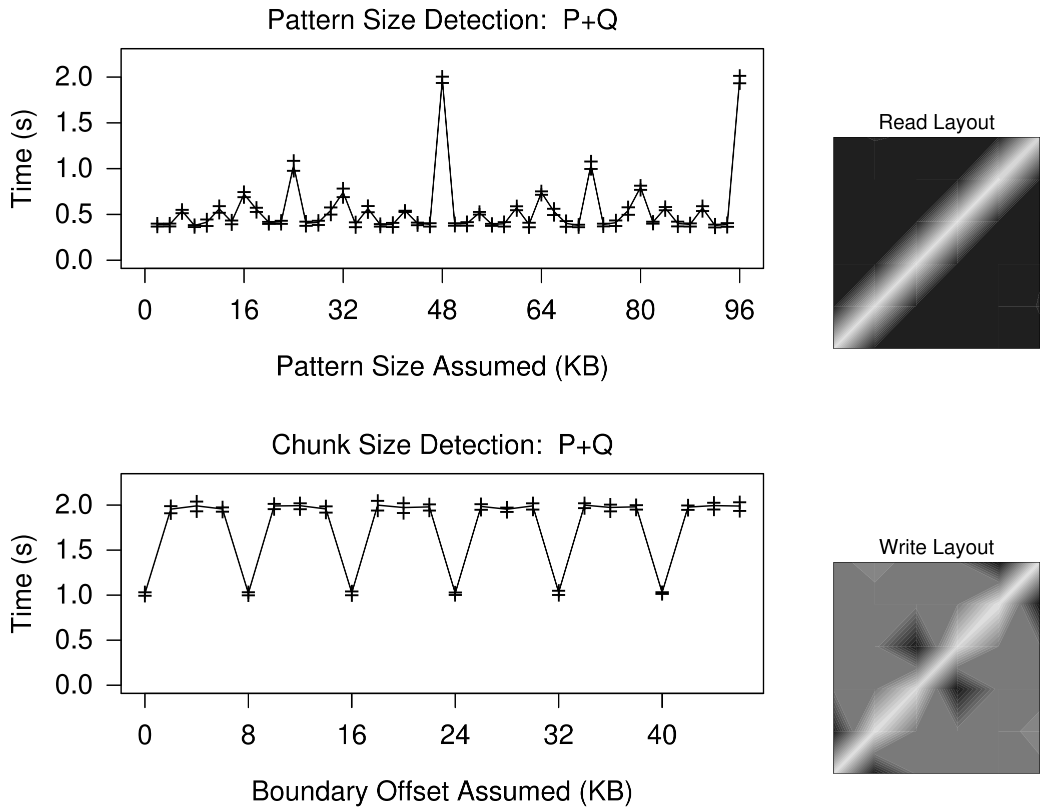


Figure 2.14 **Pattern Size, Chunk Size, and Layout Detection: P+Q.** We present simulated results for P+Q redundancy with 6 disks and a chunk size of 8 KB. The first graph confirms that the pattern size of 48 KB is detected; the second graph shows the chunk size of 8 KB is detected. The read layout graph on the right resembles RAID-0, but the write layout graph distinguishes P+Q from other schemes.

are sent to only a primary copy, then Shear will not be able to detect the presence of mirrored copies. To demonstrate that Shear handles mirroring, we consider both simple RAID-1 and chained declustering.

RAID-1: The results of running Shear on a six disk RAID-1 system are shown in Figure 2.15. Note that the pattern size in RAID-1 is exactly half of that in RAID-0, given the same chunk size and number of disks. The first graph shows how the RAID-1 pattern size of 24 KB is inferred. As Shear reads from different offsets throughout the pattern, the requests are sent to both mirrors. As desired, the worst performance occurs when the request offset is equal to the real pattern size, but in this case, the requests are serviced by two disks instead of one. This is illustrated by the fact that the worst-case time for the workload on RAID-1 is exactly half of that when on RAID-0 (*i.e.*, 1.0 instead of 2.0 seconds).

The second graph in Figure 2.15 shows how the chunk size of 8 KB is inferred. Again, as Shear tries to find the boundary between disks, requests are sent to both mirrors; Shear now automatically detects the disk boundary because the workload time increases when requests are sent to two disks instead of four disks. Since the mapping of chunks to disks within a single pattern does not contain any conflicts, the read layout and write layout graphs in Figure 2.15 resemble RAID-0.

Chained Declustering: Chained declustering [28] is a redundancy scheme in which disks are not exact mirrors of one another; instead, each disk contains a primary instance of a block as well as a copy of a block from its neighbor. The results of running Shear on a six disk system with chained declustering are shown in Figure 2.16.

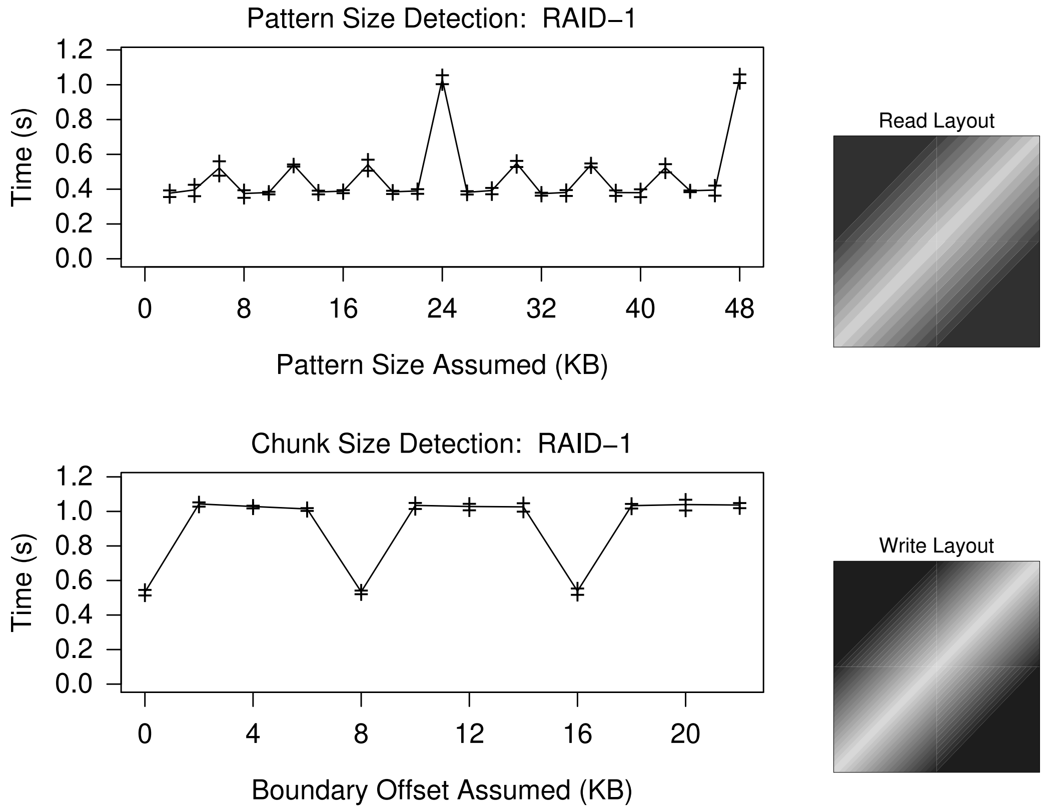


Figure 2.15 **Pattern Size, Chunk Size, and Layout Detection: RAID-1.** We present simulated results for RAID-1 with 6 disks and a chunk size of 8 KB. The first graph confirms that the pattern size of 24 KB is detected; the second graph shows the chunk size of 8 KB is detected. The read layout and write layout graphs on the right resemble those for RAID-0.

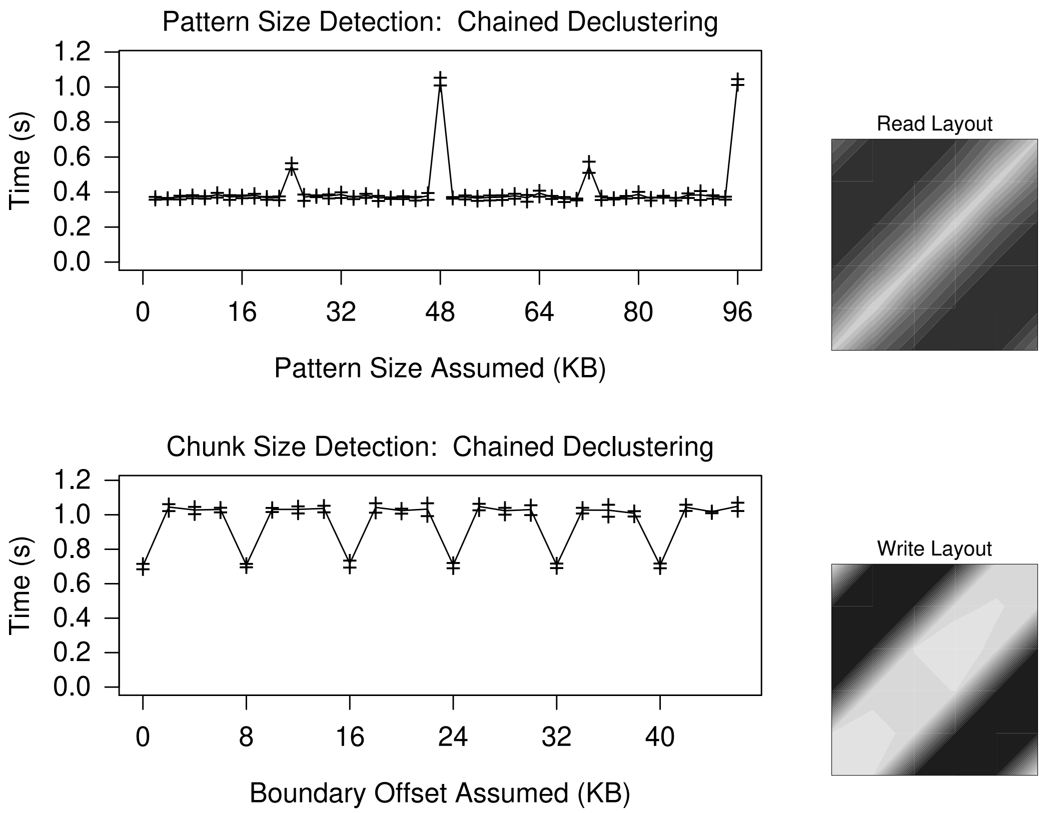


Figure 2.16 **Pattern Size, Chunk Size, and Layout Detection: Chained Declustering.** We present simulated results for chained declustering with 6 disks and a chunk size of 8 KB. The first graph confirms the pattern size of 48 KB; the second graph shows the chunk size of 8 KB is detected. The wider bands in the read layout and write layout graphs show that two neighboring chunks are mirrored across a total of three disks; this uniquely identifies chained declustering.

The first graph shows that a pattern size of 48 KB is detected, as desired. As with RAID-1, each read request can be serviced by two disks, and the pattern size is identified when all of the requests are sent to only two disks in the system. Note that the chained declustering pattern size is twice that of RAID-1 since each disk contains a unique set of data blocks.

The second graph in Figure 2.16 shows that four block chunks are again detected. However, the ratio between best and worst-case performance differs in this case from RAID-0 and RAID-1; in chained declustering the ratio is 2:3, whereas in RAID-0 and RAID-1, the ratio is 1:2. With chained declustering, when adjacent requests are located across a disk boundary, those requests are serviced by three disks (instead of four with RAID-1); when requests are located within a chunk, those requests are serviced by two disks.

The mapping conflicts with chained declustering are also interesting, as shown in the remaining graphs in Figure 2.16. With chained declustering, a pair of chunks can be located on two, three, or four disks; this results in three distinct performance regimes. This new case of three shared disks occurs for chunks that are cyclically adjacent (*e.g.*, chunks 0 and 1), resulting in the wider bands in the read and write layout graphs.

2.1.6 Overhead

We now examine the overhead of Shear, by showing how it scales as more disks are added to the system. Figure 2.17 plots the total number of I/Os that Shear generates during simulation of a variety of disk configurations. On the x-axis, we vary the configuration, and on the y-axis we plot the number of I/Os generated by the tool. Note that the RAID-5 left-asymmetric results are shown with a log scale on the y-axis.

From the graphs, we can make a few observations. First, we can see that the total number of I/Os issued for simple schemes such as RAID-0, RAID-1, and RAID-5 left-symmetric is low (in the few millions), and scales quite slowly as disks are added to the system. Thus, for these RAID schemes (and indeed, almost all others), Shear scales well to much larger arrays.

Second, we can see that when run upon RAID-5 with the left-asymmetric layout, Shear generates many more I/Os than with other redundancy schemes, and the total number of I/Os does not scale as well. The reason for this poor scaling behavior can be seen from the read layout and write layout detection bars, which account for most of the I/O traffic. As illustrated in Figure 2.1, the RAID-5 left-asymmetric pattern size grows with the square of the number of disks. Because the layout algorithms issue requests for all pairs of chunks in a pattern, large patterns lead to large numbers of requests (although many of these can be serviced in parallel); thus, RAID-5 left-asymmetric represents an extreme case for Shear. Indeed, in its current form, Shear would take roughly a few days to complete the read layout and write layout detection for RAID-5 left-asymmetric with 16 disks. However, we believe we could reduce this by a factor of ten by issuing fewer disk I/Os per pairwise trial, thus reducing run time but decreasing confidence in the layout results.

2.2 Real Platforms

In this section, we present results of applying Shear to two different real platforms. The first is the Linux software RAID device driver, and the second is an Adaptec 2200S hardware RAID

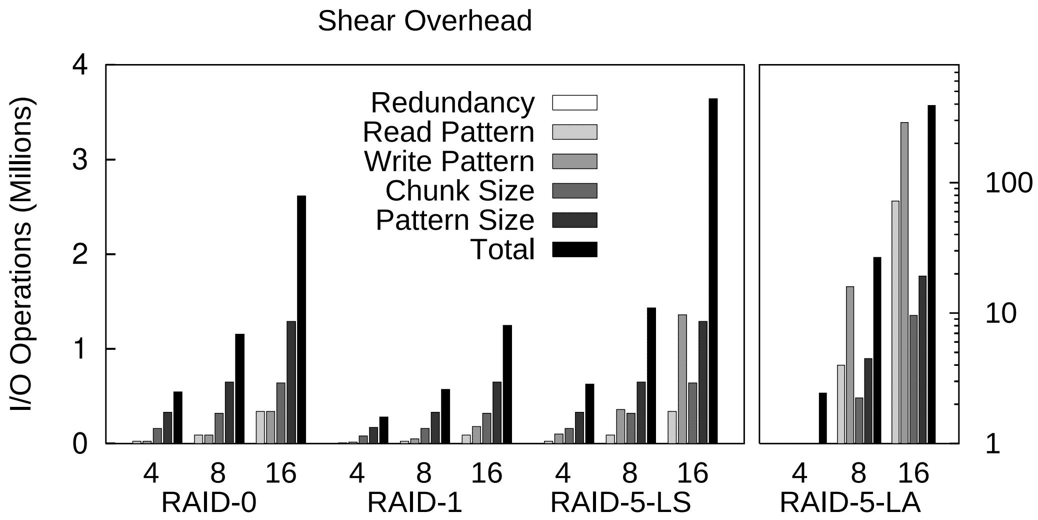


Figure 2.17 **Shear Overhead.** The graph shows the number of I/Os generated by each phase of Shear. Four simulated redundancy schemes are shown (RAID-0, RAID-1, RAID-5 left-symmetric, and RAID-5 left-asymmetric), each with three numbers of disks (4, 8, and 16) and 32 KB chunks.. Each bar plots the number of I/Os taken for a phase of Shear except the last (rightmost) bar which shows the total. The RAID-5 left-asymmetric results are plotted with a log scale on the y-axis.

controller. To understand the behavior of Shear on real systems, we ran it across a large variety of both software and hardware configurations, varying the number of disks, chunk size, and redundancy scheme. Most results were as expected; others revealed slightly surprising properties of the systems under test (*e.g.*, the RAID-5 mode of the hardware controller employs left-asymmetric parity placement). Due to space constraints, we concentrate here on the most challenging aspect of Shear: redundancy detection.

While experimenting with redundancy detection, we uncovered two issues that had to be addressed to produce a robust algorithm. The first of these was the size of the region over which the test was run. Figure 2.18 plots the read/write ratio of a single disk as the size of the region is varied.

As we can see from the figure, the size of the region over which the test is conducted can strongly influence the outcome of our tests. For example, with the Quantum disk, the desired ratio of roughly 1 is achieved only for very small region sizes, and the ratio grows to almost 2 when a few GB of the disk are used. We believe the reason for this undesirable inflation is the large settling time of the Quantum disk. Thus, we conclude that the redundancy detection algorithm should be run over as small of a portion of the disk as possible.

Unfortunately, at odds with the desire to run over a small portion of the disk is our second issue: the possible presence of a write-back cache within the RAID. The Adaptec 2200S card can be configured to perform write buffering; thus, to the host, these writes complete quickly, and are sent to the disk at some later time. Note that the presence of such a buffer can affect data integrity (*i.e.* if the buffer is non-volatile).

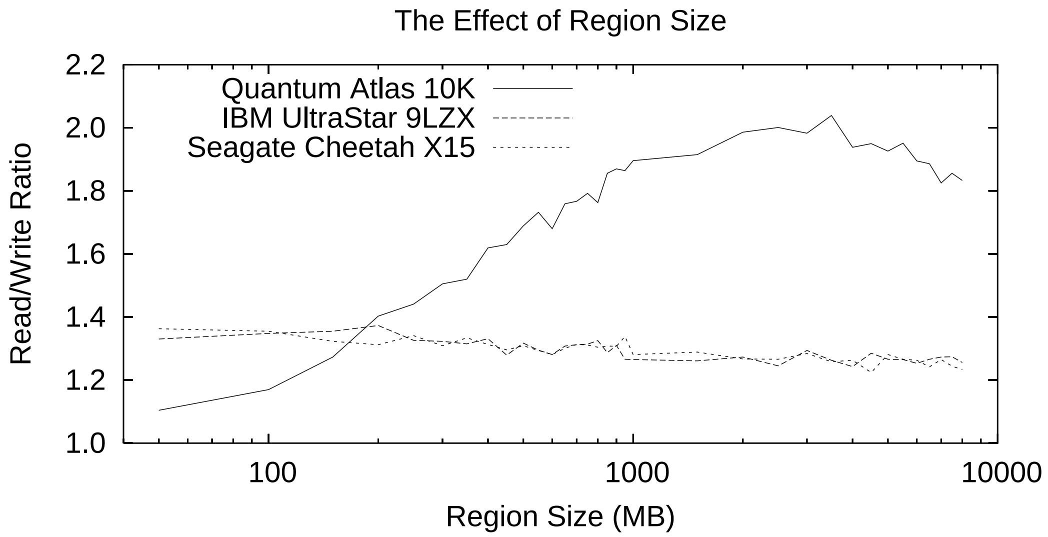


Figure 2.18 **Sensitivity to Region Size.** The figure plots the bandwidth ratio of a series of random read requests as compared to a series of random write requests. The x-axis varies the size of the region over which the experiment was run. In each run, 500 sector-sized read or write requests are issued. Lines are plotted for three different disks: a Quantum Atlas 10K 18WLS, an IBM 9LZX, and a Seagate Cheetah X15.

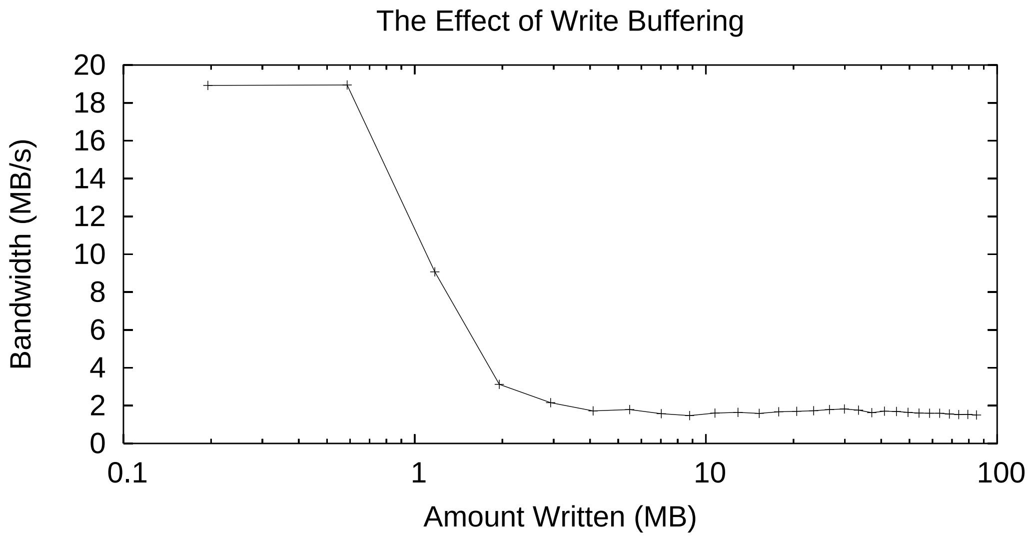


Figure 2.19 **Avoiding the Write Buffer.** The figure plots the performance of writes on top of the RAID-5 hardware with write-buffering enabled. The x-axis varies the number of writes issued, and the y-axis plots the achieved bandwidth.

Because the redundancy detection algorithm needs to issue write requests to disk to compare with read request timings, Shear must circumvent caching effects. Recall that Shear uses a simple adaptive scheme to detect and bypass buffering by issuing successive rounds of write requests and monitoring their performance. At some point, the write bandwidth decreases, indicating the RAID system has moved into the steady-state of writing data to disk instead of to memory, and thus a more reliable result can be generated. Figure 2.19 demonstrates this technique on the Adaptec hardware RAID adapter with write caching enabled.

With these enhancements in place, we study redundancy detection across both the software and hardware RAID systems. Figure 2.20 plots the read bandwidth to write bandwidth ratio across a number of different configurations. Recall that the read/write ratio is the key to differentiating the redundancy scheme that is used; for example, a ratio of 1 indicates that there is no redundancy, a

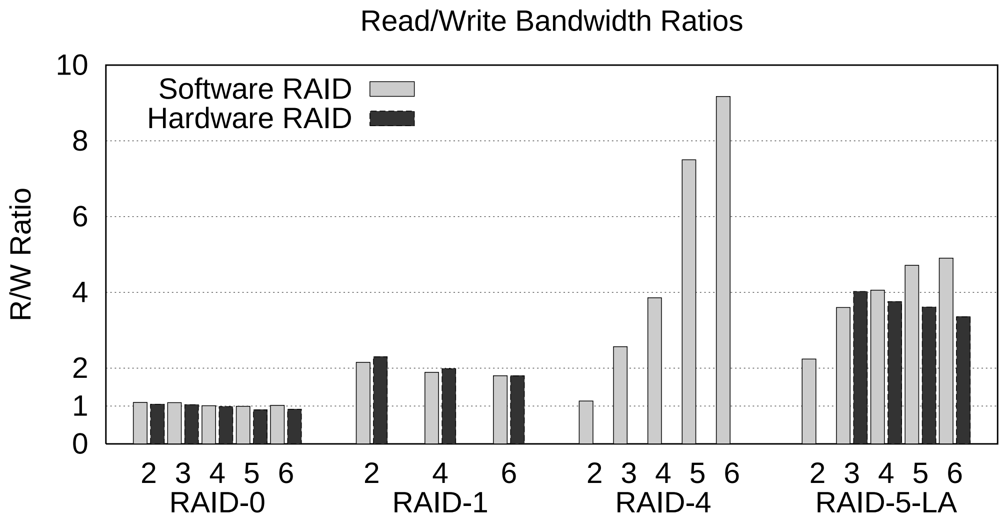


Figure 2.20 **Redundancy Detection.** The figure plots the ratio of read to write bandwidth over a variety of disk configurations. The x-axis varies the number of disks and the configuration: RAID-0, RAID-1, RAID-4, or RAID-5 left-asymmetric, with either software or hardware RAID.

ratio of 2 indicates a mirrored scheme, and a ratio of 4 indicates a RAID-5 style parity encoding. Note that our hardware RAID card does not support RAID-4 and will not configure RAID-5 on two disks.

The figure shows that Shear's redundancy detection does a good job of identifying which scheme is being used. As expected, we see read/write ratios of approximately 1 for RAID-0, near 2 for RAID-1, and 4 for RAID-5. There are a few other points to make. First, the bandwidth ratios for RAID-4 scale with the number of disks due to the parity disk bottleneck. This makes it more difficult to identify RAID-4 arrays. To do so, we rely on the write layout test described previously that exhibits this same bottleneck in write performance. The unique results from the write layout test allow us to distinguish RAID-4 from the other parity-based schemes.

Second, note the performance of software RAID-5 on 5 and 6 disks; instead of the expected read/write ratio of 4, we instead measure a ratio near 5. Tracing the disk activity and inspecting the source code revealed the cause: the Linux software RAID controller does not utilize the usual RAID-5 small write optimization of reading the old block and parity, and then writing the new block and parity. Instead, it will read in the entire stripe of old blocks and then write out the new block and parity. Finally, the graph shows how RAID-5 with 2 disks and a 2-disk mirrored system are not distinguishable; at two disks RAID-5 and mirroring converge.

2.3 Shear Applications

In this section, we illustrate a few of the benefits of using Shear. We begin by showing how Shear can be used to detect RAID configuration errors and disk failures. We then show how Shear can be used to discover information about individual disks in an array. Finally, we present an example of how the storage system parameters uncovered by Shear can be used to better tune the file system; specifically, we show how the file system can improve sequential bandwidth by writing data in full stripes.

2.3.1 Shear Management

One of our intended uses of Shear is as an administrative utility to discover configuration, performance, and safety problems. Figure 2.21 shows how a failure to identify a known scheme may suggest a storage misconfiguration. The upper set of graphs are the expected read layout graphs for the four common RAID-5 levels. The lower are the resulting read layout graphs when

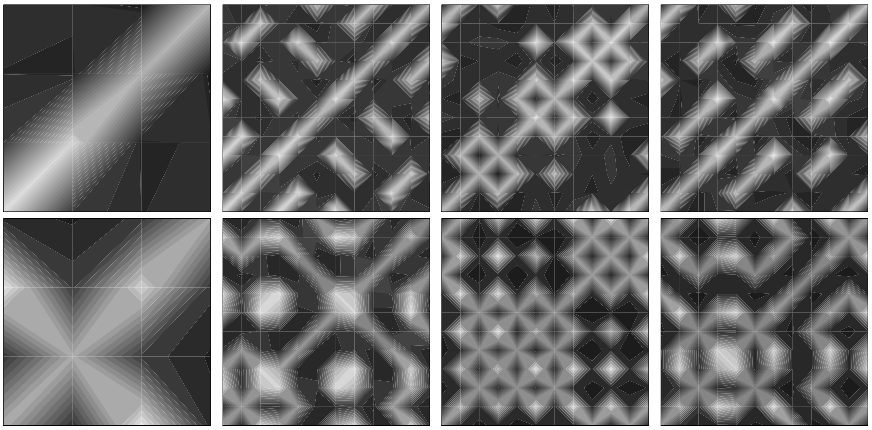


Figure 2.21 **Detecting Misconfigured Layouts.** For RAID-5 left-symmetric, left-asymmetric, right-symmetric, and right-asymmetric, the upper graph shows the read layout graph when the RAID of IBM disks is correctly configured. The lower graphs show the read layout when two logical partitions are misconfigured such that they are placed on the same physical device.

the disk array is misconfigured such that two logical partitions actually reside on the same physical disk. These graphs were generated using disk arrays comprised of four logical disks built using Linux software RAID and the IBM disks. Although the visualization makes it obvious, manual inspection is not necessary; Shear automatically determines that these results do not match any existing known schemes.

Shear can also be used to detect unexpected performance heterogeneity among disks. In this next experiment, we run Shear across a range of simulated heterogeneous disk configurations; in all experiments, one disk is either slower or faster than the rest. Figure 2.22 shows results when run upon these heterogeneous configurations.

As one can see from the figure, a faster or slower disk makes its presence known in obvious ways in both the read layout graphs as well as in the chunk size detection output (the pattern size detection is relatively unaffected). Thus, an administrator could view these outputs and clearly

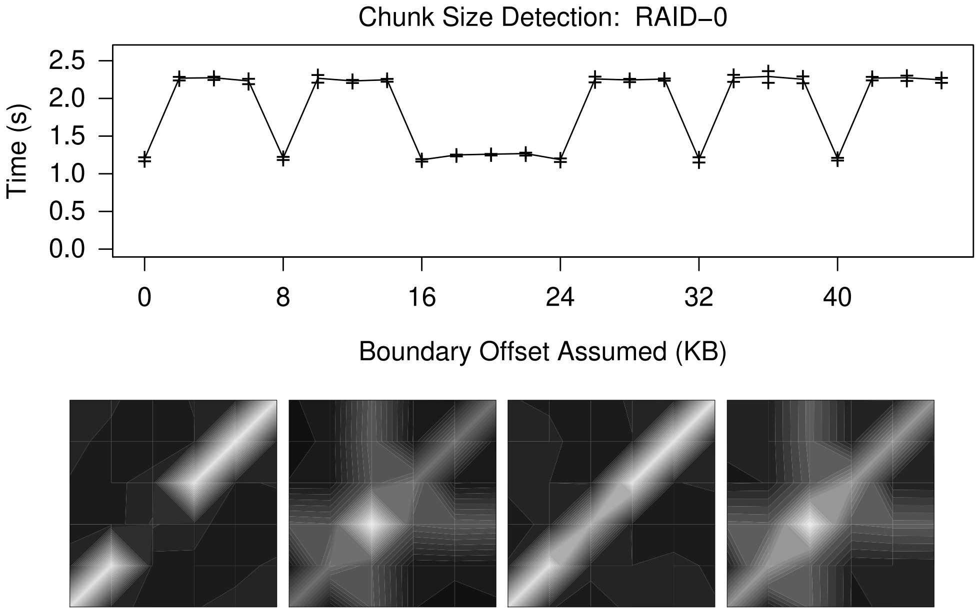


Figure 2.22 **Detecting Heterogeneity.** The first graph shows the output of the chunk size detection algorithm run upon an array with a single heterogeneous fast rotating disk. The second row of figures shows the results of the read layout algorithm on four different simulated disk configurations. In each configuration, a single disk has different capability than the others. A fast rotating, slow rotating, fast seeking, and slow seeking disk is depicted in each of the figures.

observe that there is a serious and perhaps unexpected performance differential among the disks and take action to correct the problem.

Finally, the chunk size detection algorithm in Shear can be used to identify safety hazards by determining when a redundant array is operating in degraded mode. Figure 2.23 shows the chunk size detection results for a ten disk software RAID system using the IBM disks. The upper graph shows the chunk size detection correctly working after the array was first built. The lower graph shows how chunk size detection is changed after we physically remove the fifth disk from the array. Recall that chunk size detection works by guessing possible boundaries and timing sets of requests on both sides of the boundary. Vertical downward spikes should be half the height of the plateaus and indicate that the guessed boundary is correct because the requests are serviced in parallel on two disks. The plateaus are false boundaries in which all the requests on both sides of the guessed boundary actually are incurred on just one disk. The lower graph identifies that the array is operating in degraded mode because the boundary points for the missing disk disappear, and its plateau is higher due to the extra overhead of performing on-the-fly reconstruction.

2.3.2 Shear Disk Characterization

Related projects have concentrated on extracting properties of individual disk drives [62, 76, 90]. Several techniques have been built on top of this characteristic knowledge, such as aligning files to track boundaries [63] and free-block scheduling [38]. Shear enables such optimizations in the context of storage arrays. Shear can expose the boundaries between disks, and then existing tools can be used to determine specific properties of those individual disks.

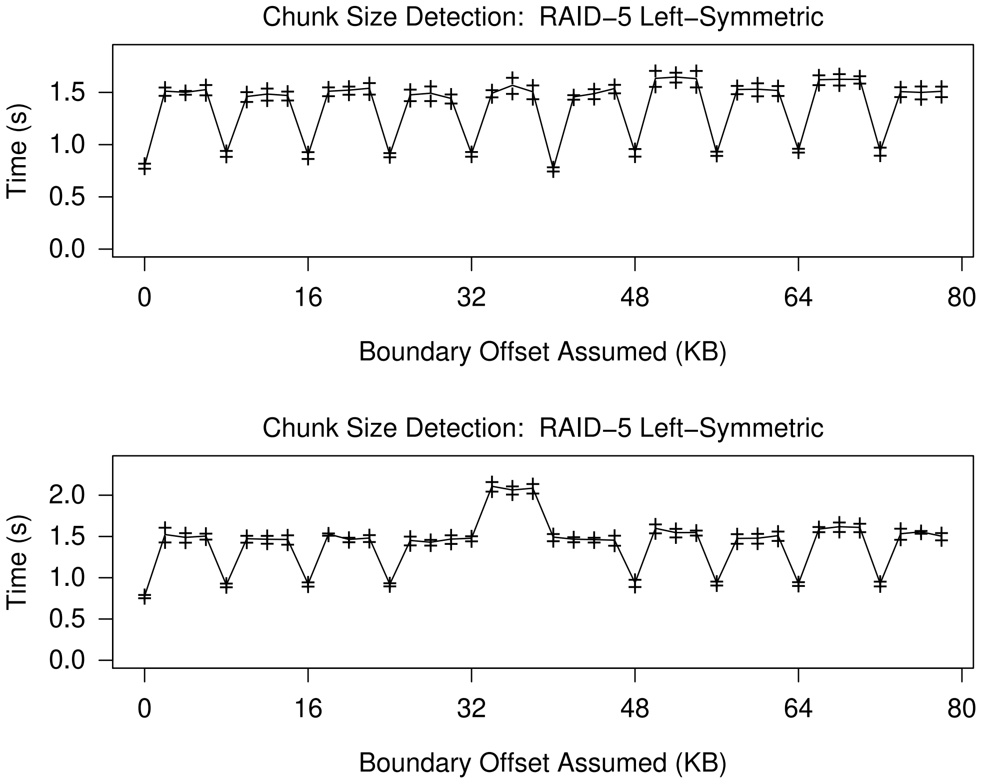


Figure 2.23 **Detecting Failure.** Using the chunk size detection algorithm, Shear can discover failed devices within a RAID system. The upper graph shows the initial chunk size detection results collected after building a 10 disk software RAID system using the IBM disks. The lower graph is for the same system after the fifth disk was removed.

We demonstrate this ability using the Skippy disk characterization tool [76]. Skippy uses a sequence of write operations at increasing strides to determine the disk sector to track ratio, rotation time, head positioning time, head switch time, cylinder switch time, and the number of recording surfaces. The first graph in Figure 2.24 shows the pattern generated by Skippy on a single Quantum disk.

The second graph in Figure 2.24 shows the results of running a modified version of Skippy on a RAID-0 array with two disks. This version of Skippy uses the array information provided by Shear to map its block reference stream to the corresponding logical blocks residing on the first disk in the array. This results in a pattern that is nearly identical to that running on a single disk, allowing us to extract the individual disk parameters. The final graph in Figure 2.24 shows the results of the same technique applied to a two disk RAID-1 array. Again, the results are nearly identical to the single disk pattern except for some small perturbations that do not affect our analysis.

There are some limitations to this approach, however. For example, in the case of RAID-1, the Skippy write workload performs as expected, but a read workload produces spurious results due to the fact that reads are balanced across disks. Conversely, reads work well under RAID-5 whereas writes do not due to the need to update parity information. Additionally, because the parity blocks under RAID-5 cannot be directly accessed, characterization tools may obtain an incomplete set of data. Despite these limitations, we have tested a read-based version of Skippy on RAID-5 and successfully extracted all parameters from the individual disks.

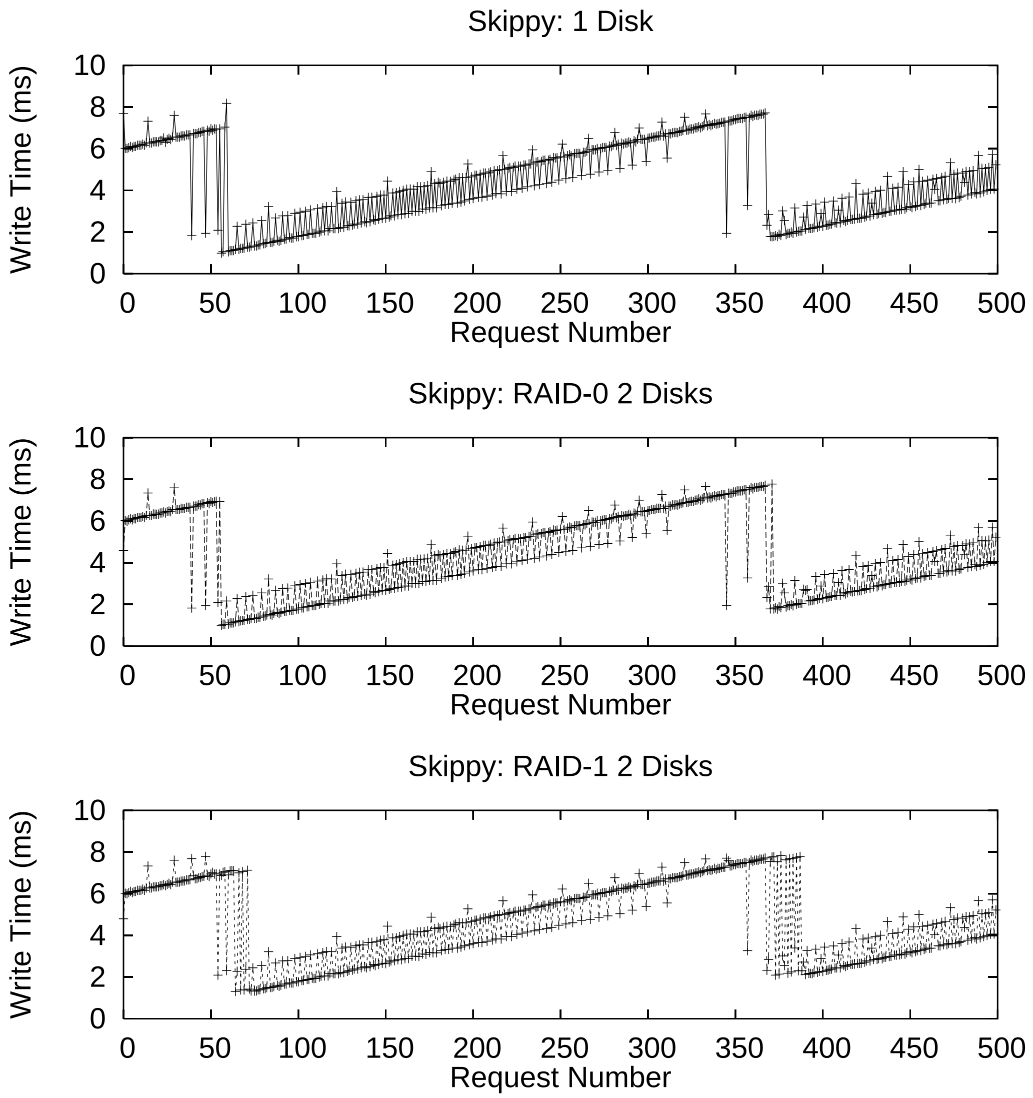


Figure 2.24 **Skippy.** The figures plot the results of running the Skippy disk characterization tool on a single Quantum disk, a two disk RAID-0 array, and a two disk RAID-1 array.

2.3.3 Shear Performance

The stripe size within a disk array can have a large impact on performance [12, 14]. This effect is especially important for RAID-5 storage, since writes of less than a complete stripe require additional I/O. Previous work has focused on selecting the optimal stripe size for a given workload. We instead show how the file system can adapt the size and alignment of its writes as a function of a given stripe size.

The basic idea is that the file system should adjust its writes to be stripe aligned as much as possible. This optimization can occur in multiple places; we have modified the Linux 2.4 device scheduler so that it properly coalesces and/or divides individual requests such that they are sent to the RAID in stripe-sized units. This modification is straight-forward: only about 20 lines of code were added to the kernel.

This simple change to make the file system stripe-aware leads to tremendous performance improvements. The experiments shown in Figure 2.25 are run on a hardware RAID-5 configuration with four Quantum disks and a 16 KB chunk size. These results show that a stripe-aware file system noticeably improves bandwidth for moderately-sized files and improves bandwidth for larger files by over a factor of two.

2.4 Discussion

Our approach to uncovering the details of a storage array is not without its weaknesses. First, the requirement of homogeneous disks limits the scope of systems that Shear can successfully examine. The key to overcoming this limitation lies in determining the pattern size over a set of

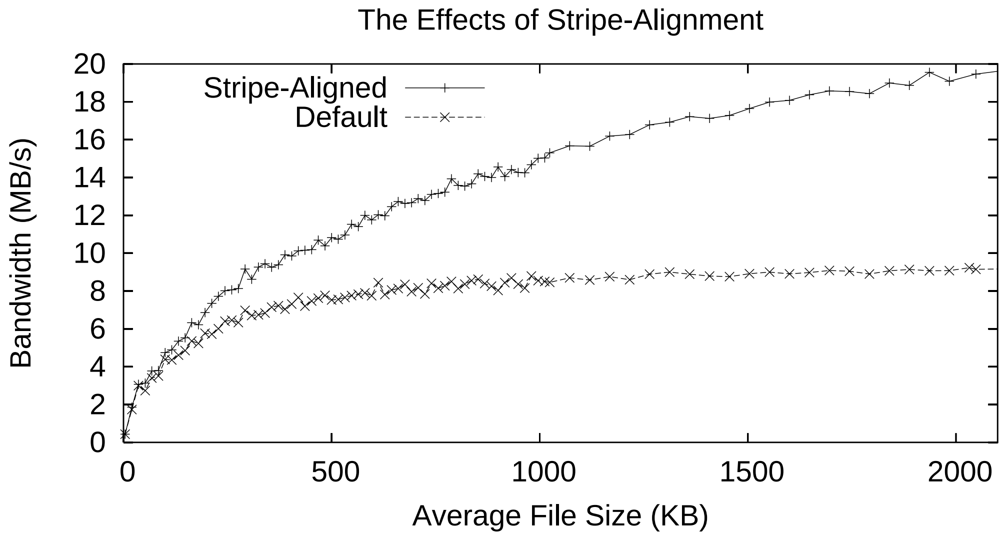


Figure 2.25 **Benefits of Stripe Alignment.** The figure plots the bandwidth of a series of file creations of an average size, as varied along the x-axis. Two variants are shown: one in which the file system generates stripe-sized writes and the default Linux. The workload consists of creating 100 files. The x-axis indicates the mean size of the files, which are uniformly distributed between $0.5 \times$ mean and $1.5 \times$ mean.

heterogeneous disks. We believe the same algorithmic approach can be utilized, but the trials conducted may need to be deterministic, and the analysis phase may require modifications to establish the performance differences of the array components. Second, the Shear detection process may take a long time depending on the size and particular layout of the array. It may be possible to use fewer requests during the individual microbenchmarks to reduce this runtime, but we have not explored the sensitivity of our analysis to this parameter. Finally, Shear requires that it is the only process accessing the array, and this prohibits the testing of storage systems that cannot be taken offline. In the future, it may be possible to position Shear to augment an existing workload to induce the desired microbenchmarks in an online system, though doing so without severe detriment to foreground performance will be challenging.

2.5 Conclusions

In this chapter we have presented Shear, a system that automatically detects important characteristics of modern storage arrays, including the number of disks, chunk size, level of redundancy, and layout scheme. The keys to Shear are its use of randomness to extract steady-state performance and its use of statistical techniques to deliver automated and reliable detection. We have verified that Shear works as desired through a series of simulations over a variety of layout and redundancy schemes. We have subsequently applied Shear to both software and hardware RAID systems, revealing properties of both. Specifically, we found that Linux software RAID exhibits poor performance for RAID-5 parity updates, and the Adaptec 2200S RAID adapter implements RAID-5 left-asymmetric layout.

We have also shown how Shear could be used through three case studies. Storage administrators can use Shear to verify properties of their storage arrays, monitor their performance, and detect disk failures. Shear can help extract individual parameters from disks within an array, enabling performance enhancements previously limited to single disk systems. Finally, we have shown a factor of two improvement in performance from a file system tuning its writes to the stripe size of its RAID storage.

Chapter 3

Bridging the Information Gap: Exposed RAID and Informed LFS

Although our basic informing interface has shown to be useful, it provides details at a rather low level. File systems that want to take advantage of the array configuration must be imbued with particular knowledge of each possible RAID scheme and its unique performance and reliability characteristics. Given the number of RAID variants that exist today, and the potential growth of new schemes in the future, designing a file system that can account for such a large population may prove difficult.

To overcome this limitation, we introduce a second informing interface, Exposed RAID, that encapsulates array information in abstractions that are meaningful to file system objectives. Specifically, the E×RAID address space is divided into a set of regions, each of which is mapped to a single disk or a set of disks. Hence, these regions represent the performance and failure boundaries within the disk array. In addition to this static information, E×RAID provides dynamic information about the performance and reliability of each region that may be exploited by the file system to manage its use of the storage.

We make use of the E$\times$RAID informing interface to evaluate a new division of labor between the storage system and the file system. In particular, we design an Informed Log-Structured File System (I·LFS) that explicitly manages and takes advantage of the performance and failure boundaries present in a multiple disk storage system. By combining the information provided by E$\times$RAID along with file-system specific knowledge, I·LFS is more flexible and manageable than a traditional file system, and can deliver higher performance and availability as well. For example, adding a disk to I·LFS on-line is easily accomplished; further, I·LFS accounts for the potential heterogeneity introduced by a new disk, and dynamically balances load across the disks of the system, whatever their rates. I·LFS also increases the flexibility of storage by enabling user control over redundancy on a per-file basis, and implements lazy mirroring to defer replication to a later time, potentially increasing performance of the system at a slight decrease in reliability. Crucial to I·LFS/E$\times$RAID is the implementation of the aforementioned benefits without a significant increase in overall complexity (and thus maintainability) of the storage stack. Via careful design, all the functionality mentioned above is implemented with only a 19% increase in overall code size as compared to a traditional system.

The rest of the chapter is structured as follows. We give an overview of our approach in Section 3.1, and then we describe E$\times$RAID and I·LFS in Sections 3.2 and 3.3, respectively. We present an evaluation of our system in Section 3.4, a discussion in Section 3.5, and we conclude in Section 3.6.

3.1 Overview

In the next two sections, we present the design and implementation of E×RAID and I·LFS. Our primary goal in designing the system is to exploit the information made available by E×RAID, thus allowing I·LFS to implement functionality that would be difficult to achieve in a more traditional layering. In particular, we aim to increase: (1) the ease of storage management, (2) performance, especially when considering multiple heterogeneous disks, and (3) functionality, so as to meet the demands of a diverse set of applications.

Our primary goal in designing and implementing E×RAID is to facilitate the use of array information in the most efficient and simplest possible way, and to allow non-informed legacy file systems to be built on top of E×RAID with no changes. Our primary goal in implementing I·LFS is to minimize the impact of transforming the file system to utilize the new storage interface. For example, changes that would require a re-design of the vnode layer were ruled out, as that would mandate that all other file systems be changed in order to function in our system. Thus, throughout our implementation effort, we integrate changes into I·LFS in a highly localized and modular fashion – the fewer lines of code that changed, the better.

One question that must be addressed is our decision to modify LFS and not a more traditional (or perhaps more popular) FFS-like or journaling file system. One reason we chose LFS is its natural flexibility in data placement; LFS is a modern example of a "write anywhere" storage system [27, 32]. Write-anywhere systems provide an extra level of indirection such that writes can be placed in any location on the storage medium, and we exploit this aspect of LFS in part of our implementation. However, with this in mind, we do believe that a number of our implementation techniques are general and could be applied to other file systems, and hope to investigate doing so in the future. Those interested in general LFS file system performance issues should consult the work of Rosenblum and Ousterhout [55], or subsequent research by Seltzer *et al.* [65, 66].

All of our software was developed within the context of the NetBSD 1.5 operating system. The E×RAID layer was implemented as a set of hooks on the lower-level block-driver calls, and is described in more detail in Section 3.2. I·LFS was implemented by extending the NetBSD version of LFS, which is based on the original LFS for BSD Unix [65], and is described in detail in Section 3.3. We chose the NetBSD version of LFS as it is known to be a relatively stable and solid implementation.

3.2 Exposed RAID

We now describe the Exposed RAID storage interface. It consists of two major components: a segmented address space which exposes some or all of the parallelism of the storage system to the file system, and functions used to inform the file system of the dynamic state of the storage system.

The purpose of the E×RAID layer is to encapsulate information about a storage array in abstractions that are meaningful to the file system. Thus, the file system, its designers, and its implementors are insulated from the particular implementation of the array, and can focus on notions of functionality, performance, and reliability.

The realization of this abstraction may occur in a number of ways. For instance, a storage system could provide native support for the E×RAID interface. Alternatively, the interface could be implemented as a software layer on top of our basic informing interface, performing block remapping to provide E$\times$RAID regions and dynamic monitoring of performance. Additionally, Shear could be used to automatically determine the parameters of the basic informing interface if the underlying array is a legacy system.

3.2.1 A Segmented Address Space

A traditional RAID array presents the storage subsystem to the file system as a linear array of blocks, underneath of which the true complexity of the particular RAID scheme is hidden. File systems interact with RAID systems by either reading or writing the blocks. In keeping with our desire to minimize change and preserve backwards compatibility, E$\times$RAID also provides a linear array of blocks which can be read or written as the basic interface.

However, because we wish to expose information about the storage system to the file system, the address space is *segmented*; specifically, it is organized as a series of contiguous *regions*, each of which is mapped directly to a single disk (or set of disks), and these region boundaries are made known to the file system above, if it so desires. For example, in a four-disk storage system with each disk capable of storing N blocks, the address space E$\times$RAID presents might be segmented as follows: blocks 0 through $N-1$ map to disk 0, blocks N through $2N-1$ map to disk 1, and so forth.

By exposing this information, E$\times$RAID enables the file system to understand the performance and failure boundaries of the storage system. As we shall see in later sections, the file system can take advantage of this to place data on a particular region more intelligently, potentially improving performance, reliability, or other aspects of the storage system.

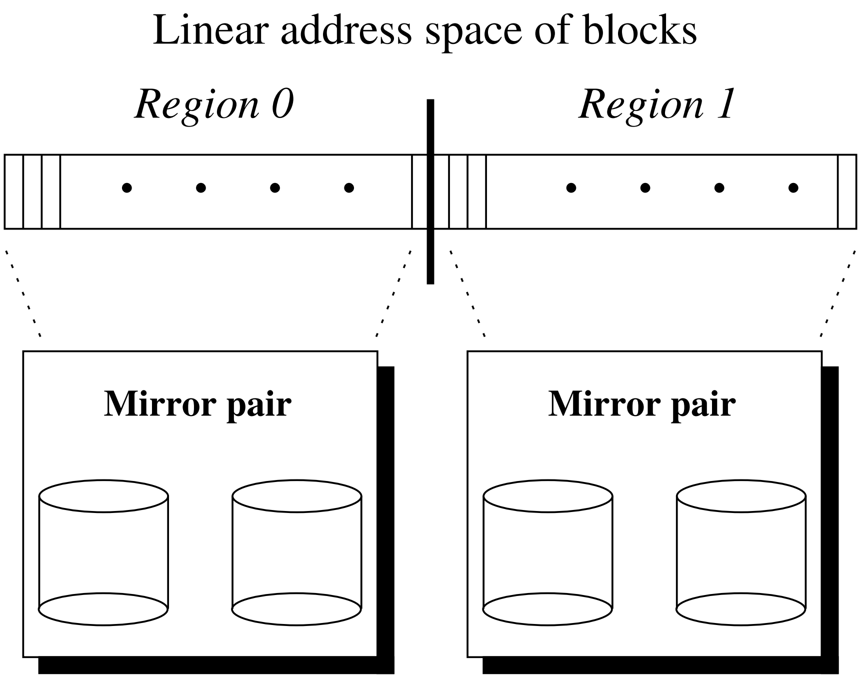


Figure 3.1 **An Example E×RAID Configuration.** The diagram depicts an example E×RAID configuration in which each of two disks is combined into a mirrored pair. Two regions, each half of the size of the total address space, are presented to the client file system.

Within E×RAID, a region may represent more than just a single disk. For example, a region could be configured to represent a mirrored pair of disks, or even a RAID-5 collection. Thus, each region can be viewed as a configurable software-based RAID, and the entire E×RAID address space as a single representation of the conglomeration of such RAID subsystems. In such a scenario, some information is hidden from the file system, but cross-region optimizations are still possible, if more than one region exists. An example of an E×RAID configuration over mirrored pairs is shown in Figure 3.1.

Allowing each region to represent more than just a single disk has two primary benefits. First, if each region is configured as a RAID (such as a mirrored pair of disks), the file system is not forced to manage redundancy itself, though it can choose to do so if so desired. Second, this arrangement allows for backwards compatibility, as E×RAID can be configured as a single striped, mirrored, or RAID-5 region, thus allowing unmodified file systems to use it without change.

3.2.2 Dynamic Information

Although the segmented address space exposes the nature of the underlying disk system to the file system (either in part or in full), this knowledge is often not enough to make intelligent decisions about data placement or replication. Thus, the E×RAID layer exposes dynamic information about the state of each region to the file system above, and it is in this way that E×RAID distinguishes itself from traditional volume managers.

Two pieces of information are needed. First, the file system may desire to have *performance* information on a per-region basis. The E×RAID layer tracks queue lengths and current throughput levels, and makes these pieces of information available to the file system. Historical tracking of information is left to the file system.

Second, the file system may wish to know about the resilience of each region, *i.e.*, when failures occur, and how many more failures a region can tolerate. Thus, E×RAID also presents this information to the file system. For example, in Figure 3.1, the file system would know that each mirror pair could tolerate a single disk failure, and would be informed when such a failure occurs. The file system could then take action, perhaps by directing subsequent writes to other regions, or even by moving important data from the "bad" region into other, more reliable portions of the E×RAID address space.

3.2.3 Implementation

In our current implementation, E×RAID is implemented as a thin layer between the file system and the storage system. In order to implement a striped, mirrored, or RAID-5 region, we simply utilize the standard software RAID layer provided with NetBSD. However, our prototype E×RAID layer is not completely generalized as of this date, and thus in its current form would require some effort to allow a file system other than I·LFS to utilize it.

The segmented address space is built by interposing on the vnode *strategy* call, which allows us to remap requests from their logical block number within the virtual address space presented by E×RAID into a physical disk number and block offset, which can then be issued to underlying disk or RAID.

Dynamic performance information is collected by monitoring the current performance levels of reads and writes. In the prototype, region boundaries, failure information, and performance levels (throughput and queue length) are tracked in the low-levels of the file system. A more complete implementation would make the information available through an `ioctl()` interface to the E×RAID device. Also note that we focus primarily on utilizing the performance information in this chapter.

3.3 Informed LFS

We now describe the I·LFS file system. Our current design has four major pieces of additional functionality, as compared to the standard LFS: on-line expandability of the storage system, dynamic parallelism to account for performance heterogeneity, flexible user-managed redundancy, and lazy mirroring of writes. In sum total, these added features make the system more manageable (the administrator can easily add a new disk, without worry of configuration), more flexible (users have control over if replication occurs), and have higher performance (I·LFS delivers the full bandwidth of the system even in heterogeneous configurations, and flexible mirroring avoids some of the costs of more rigid redundancy schemes). For most of the discussion, we focus on the case that most separates I·LFS/E×RAID from a traditional RAID, where the E×RAID layer exposes each disk of the storage system as a separate region to I·LFS.

3.3.1 On-Line Expansion and Contraction

Design: The ability to upgrade a storage system incrementally is crucial. As the performance or capacity demands of a site increase, an administrator may need to add more disks. Ideally, such an addition should be simple to perform (*e.g.*, a single command issued by the administrator, or an automatic addition when the disk is detected by the hardware), require no down-time (thus keeping availability of storage high), and immediately make the extra performance and capacity of the new disk available.

In older systems, on-line expansion is not possible. Even if the storage system could add a new disk on-the-fly, it is likely the case that an administrator would have to unmount the partition, expand it (perhaps with a tool similar to that described in [79]), and then re-mount the file system. Worse, some systems require that a new file system be built, forcing the administrator to restore data from tape. More modern volume managers [85] allow for on-line expansion, but still need file system support.

Thus, our I·LFS design includes the ability to incorporate new disks (really, new E×RAID regions) on-line with a single command given to the file system. No complicated support is necessitated across many layers of the system. If the hardware supports hot-plug and detection of new disks without a power-cycle, I·LFS can add new disks without any down time and thus reduction in data availability. Overall, the amount of work an administrator must put forth to expand the system is quite small.

Contraction is also important, as the removal of a region should be as simple as the addition of one. Therefore, we also incorporate the ability to remove a region on the fly. Of course, if the file system has been configured in a non-redundant manner, some data will likely be lost. The difference between I·LFS and a traditional system in this scenario is that I·LFS knows exactly which files are available and can deliver them to applications.

Implementation: To allow for on-line expansion and contraction of storage, the file system views regions that have not yet been added as extant and yet fully utilized; thus, when a new region is added to the system, the blocks of that disk are made available for allocation, and the file system will immediately begin to write data to them. Conversely, a region that is removed is viewed as fully allocated. This technique is general and could be applied to other file systems, and similar ideas have been used elsewhere [27].

More specifically, because a log-structured file system is composed of a collection of LFS segments, it is natural to expand capacity within I·LFS by adding more free segments. To implement this functionality, the `newfs_ilfs` program creates an expanded LFS segment table for the file system. The entries in the segment table record the current state of each segment. When a new E×RAID region is added to the file system, the pertinent information is added to the superblock, and an additional portion of the segment table is activated. This approach limits the number of regions that can be added to a fixed number (currently, 16); for more flexible growth, the segment table could be placed in its own file and expanded as necessary.

3.3.2 Dynamic Parallelism

Design: One problem introduced by the flexibility an administrator has in growing a system is the increased potential for performance heterogeneity in the disk subsystem; in particular, a new disk or E×RAID segment may have different performance characteristics than the other disks of the system. In such a case, traditional striping and RAID schemes do not work well, as they all assume that disks run at identical rates [5, 19].

Traditionally, the presence of multiple disks is hidden by the storage layer from the file system. Thus, current systems must handle any disk performance heterogeneity in the storage layer – the file system does not have enough information to do so itself. The research community has proposed schemes to deal with static disk heterogeneity [4, 19, 59, 91], though many of these solutions require careful tuning by an administrator. As Van Jacobsen notes, “Experience shows that anything that needs to be configured will be misconfigured” [30].

Further complicating the issue is that the delivered performance of a device could change over time. Such changes could result from workload imbalances, or perhaps from the “fail-stutter” nature of modern devices, which may present correct operation but degraded performance to clients [6]. Even if more advanced heterogeneous data layout schemes are utilized, they will not work well under dynamic shifts in performance.

To handle such static and dynamic performance differences among disks, we include a dynamic segment placement mechanism within I·LFS [5]. A segment can logically be written to any free space in the file system; we exploit this by writing segments to E×RAID regions in proportion to their current rate of performance, exploiting the dynamic state presented to the file system by E×RAID. By doing so, we can dynamically balance the write load of the system to account for static or dynamic heterogeneity in the disk subsystem. Note that if performance of the disks is roughly equivalent, this dynamic scheme will degenerate to standard RAID-0 striping of segments across disks.

This style of dynamic placement could also be performed in a more traditional storage system (*e.g.*, AutoRAID has the basic mechanisms in place to do so [88]). However, doing so unduly adds complexity into the system, as *both* the file system and the storage system have to track where blocks are placed; by pushing dynamic segment placement into the file system, overall complexity is reduced, as the file system already tracks where the blocks of a file are located.

Implementation: The original version of LFS allocates segments sequentially based on availability; in other words, all free segments are treated equally. To better manage parallelism among disks in I·LFS, we develop a *segment indirection* technique. Specifically, we modify the `ilfs_newseg()` routine to invoke a data placement strategy. The `ilfs_newseg()` routine is used to find the next free segment to write to; here, we alter it to be "region aware", and thus allow for a more informed segment-placement decision. By choosing disks in accordance with their performance levels (information provided by E×RAID), the load across a set of heterogeneously-performing regions can be balanced.

The major advantage of our decision to implement this functionality within the `ilfs_newseg()` routine is that it localizes the knowledge of multiple disks to a very small portion of the file system; the vast majority of code in the file system is not aware of the region boundaries within the disk address space, and thus remains unchanged. The slight drawback is that the decision of which region to place a segment upon is made early, before the segment has been written to; if the performance level of the disk changes as the segment fills in a significant way, the placement decision could potentially be a poor one. In practice, we have not found this to be a performance problem.

3.3.3 Flexible Redundancy

Design: Typically, redundancy is implemented in a one-size-fits-all manner, as a single RAID scheme (or two, as in AutoRAID) is applied to all the blocks of the storage system. The file system is typically neither involved nor aware of the details of data replication within the storage layer. This traditional approach is limiting, as much semantic information is available in the file system as well as in smart users or applications, which could be exploited to improve performance or better utilize capacity.

Thus, in I·LFS, we explore the management of redundancy strictly within the file system, as managing redundancy in the file system provides greater flexibility and control to users. In our current design, we allow users or applications to select whether a file should be made redundant (in particular, if it should be mirrored). If a file is mirrored, users pay the cost in terms of performance and capacity. If a file is not mirrored, performance increases during writes to that file, and capacity is saved, but the chances of losing the file are increased. Turning off redundancy is thus well-suited for temporary files, files that can easily be regenerated, or swap files.

Because I·LFS performs the replication, better accounting is also possible, as the system knows exactly which files (and hence which users) are using which physical blocks. In contrast, with a traditional file system mounted on top of an advanced storage system such as AutoRAID [88], users are charged based on the logical capacity they are using, whereas the true usage of storage depends on access patterns and usage frequency.

Because redundancy schemes are usually implemented within the RAID storage system (where no notion of a file exists), our scheme would not easily be implemented in a traditionally-layered system. The storage system is wholly unaware of which blocks constitute a file and therefore cannot receive input from a user as to which blocks to replicate; only if both the file system and storage system were altered could such functionality be realized. In the future, it would be interesting to investigate a range of policies on top of our redundancy mechanisms that automatically apply different redundancy strategies according to the class of a file, akin to how the Elephant file system segregates files for different versioning techniques [60].

Implementation: To accomplish our goal of per-file redundancy, we decided to utilize separate and unique meta-data for original and redundant files. This approach is natural within the file system as it does not require changes to on-disk data structures.

In our implementation, we use a straight-forward scheme that assigns even inode numbers to original files and odd inode numbers to their redundant copies. This method has several advantages. Because the original and redundant files have unique inodes, the data blocks can be distributed arbitrarily across disks (given certain constraints described below), thus allowing us to use redundancy in combination with our other file system features. Also, the number of LFS inodes is unlimited because they are written to the log, and the inode map is stored in a regular file which is expanded as necessary. The prime disadvantage of our approach is that it limits redundancy to one

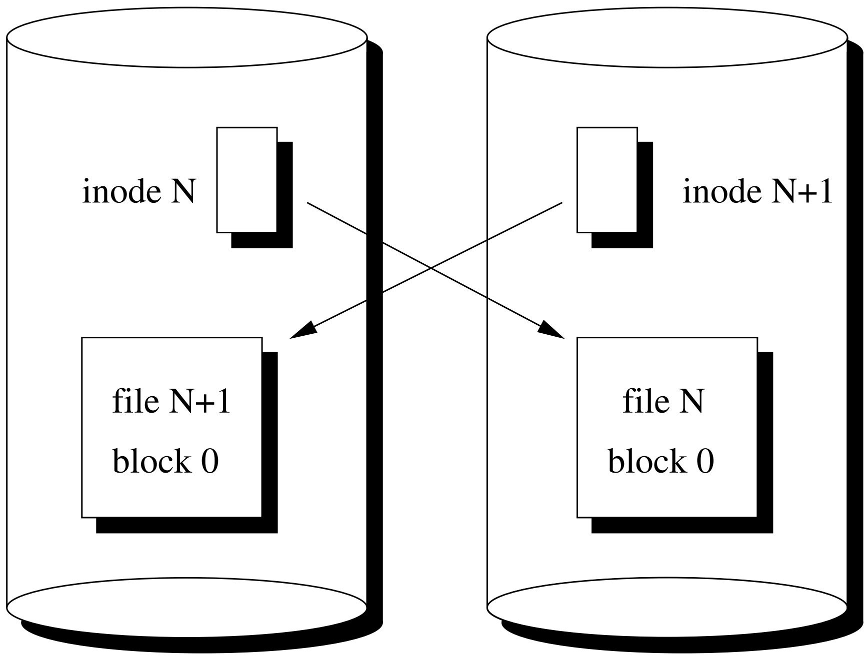


Figure 3.2 **The Crossed Pointer Problem.** The figure illustrates the problem with using a separate file as a means for redundancy; specifically, even though each element of a file (inode, data block) has been replicated, a single lost disk could still make it difficult to find a particular data block, due to the extra requirement that for each block, a pointer chain to the block must still be live. In the example, the file with inode number N and its mirror, inode $N + 1$, consist of a single data block (block 0). If either disk crashes, it is not possible to find the corresponding data block, even though a copy of it exists on the remaining working disk.

copy, but this could easily be extended to an N-way mirroring scheme by reserving N i-numbers per file.

One problem introduced by our decision to utilize separate inodes to track the primary and mirrored copy of a file is what we refer to as the crossed pointer problem. Figure 3.2 illustrates the difficulty that can arise. Simply requiring each component of a file (*e.g.*, the inode, indirect blocks, and data blocks) be replicated is not sufficient to guarantee that all data can be recovered easily under a single disk failure. Instead, we must ensure that each data block is *reachable* under a disk failure; a block being reachable implies that a pointer chain to it exists.

Consider the example in the figure: a file with inode number N is replicated within inode number $N + 1$. Inode N is located on the first disk, as is the first data block of the mirror copy (file $N + 1$). Inode $N + 1$ is on the other disk, as is the first data block of the primary copy (file N). However, if either disk fails, the first data block is not easily recovered, as the inode on the surviving disk points to the data block on the failed disk. In some file systems, this would be a fatal flaw, as the data block would be unrecoverable. In LFS, it is only a performance issue, as the extra information found within segment summary blocks allows for full recovery; however, a disk crash would mandate a full scan of the disk to recover all data blocks.

There are a number of possible remedies to the problem. For example, one could perform an explicit replication of each inode and all other pointer-carrying structures, such as indirect blocks, doubly-indirect blocks, and so forth. However, this would require the on-disk format to change, and would be inefficient in its usage of disk space, as each inode and indirect block would have four logical copies in the file system.

Instead, we take a much simpler approach of *divide and conquer*. The disks of the system are divided into two sets. When writing a redundant file to disk, I·LFS decides which set the primary copy should be placed within; the redundant copy is placed within the other set. Thus, because no pointers cross from either set into the other, we can guarantee that a single failure will cause no harm (in fact, we can tolerate any number of failures to disks in that set).

Finally, incorporating redundancy into I·LFS also presents us with a difficult implementation challenge: how should we replicate the data and inodes within the file system, without re-writing every routine that creates or modifies data on disk? We develop and apply *recursive vnode invocation* to ease the task. We embellish most I·LFS vnode operations with a short recursive tail; therein, the routine is invoked recursively (with appropriate arguments) if the routine is currently operating on an even i-number and therefore on the primary copy of the data, and if the file is designated for redundancy by the user. For instance, when a file is created using `ilfs_create()`, a recursive call to `ilfs_create()` is used to create a redundant file. The recursion is broken within the call to perform the identical operation to the redundant file.

3.3.4 Lazy Mirroring

Design: User-controlled replication allows users to control *if* replication occurs, but not *when*. As has been shown in previous work, many potential benefits arise in allowing flexible control over when redundant copies are made or parity is updated [18]. Delaying parity updates has been shown to be beneficial in RAID-5 schemes to avoid the small-write problem [61], and could also reduce load under mirrored schemes. Implementing such a feature at the file system level allows the user to decide the window of vulnerability for each file, as losing data in certain files may likely be more tolerable than in others. Note that either of these enhancements would be difficult to implement in a traditional system, as the information required resides in both the file system and RAID, necessitating non-trivial changes to both.

In I·LFS, we incorporate *lazy mirroring* into our user-controlled replication scheme. Thus, users can designate a file as non-replicated, immediately replicated, or lazily replicated. By choosing a lazy replica, the user is willing to increase the chance of data loss for improved performance. Lazy mirroring can improve performance for one of two reasons. First, by delaying file replication, the file system may reduce load under a burst of traffic and defer the work of replication to a later period of lower system load. Second, if a file is written to disk and then deleted before the replication occurs, the cost of replication is removed entirely. As most systems buffer files in memory for a short period of time (*e.g.*, 30 seconds), and file lifetimes have recently been shown to be longer than this on average [53], this second scenario may be more common than previously thought.

Implementation: Lazy mirroring is implemented in I·LFS as an embellishment to the file-system cleaner. For files that are designated as lazy replicas, an extra bit is set in the segment usage table indicating their status. When the cleaner scans a segment and finds blocks that need to be replicated, it simply performs the replication directly, making sure to place replicated blocks so as to avoid the crossed pointer problem, and associates them with the mirrored inode. When the replication is complete, the bit is cleared. Currently, the file system replicates files after a 2-minute delay, though in the future this could be set directly by the user or application.

3.4 Evaluation

In this section, we present an evaluation of E×RAID and I·LFS. Experiments are performed upon an Intel-based PC with 128 MB of physical memory. The main processor is a 1-GHz Intel Pentium III Xeon, and the system houses four 10,000 RPM Seagate ST318305LC Cheetah 36XL disks (which we will refer to as the "fast" disks), and four 7,200 RPM Seagate ST34572W Barracuda 4XL disks (the "slow" disks). The fast disks can deliver data at roughly 21.6 MB/s each, and the slow disks at approximately 7.5 MB/s apiece. For all experiments, we perform 30 trials and show both the average and standard deviation.

In some experiments, we compare the performance of I·LFS/E×RAID to standard RAID-0 striping. Stripe sizes are chosen so as to maximize performance of the RAID-0 given the workload at hand, making the comparison as fair as possible, or even slightly unfair towards I·LFS/E×RAID.

3.4.1 Baseline Performance

In this first experiment, we demonstrate the baseline performance of I·LFS and E×RAID on top of two different homogeneous storage configurations, one with four slow disks, and one with four fast disks. The experiment consists of sequential write, sequential read, random write, and random read phases (based on patterns generated by the Bonnie [9] and IOzone [43] benchmarks). We perform this experiment to demonstrate that there is no unexpected overhead in our implementation, and that it scales to higher-performance disks effectively.

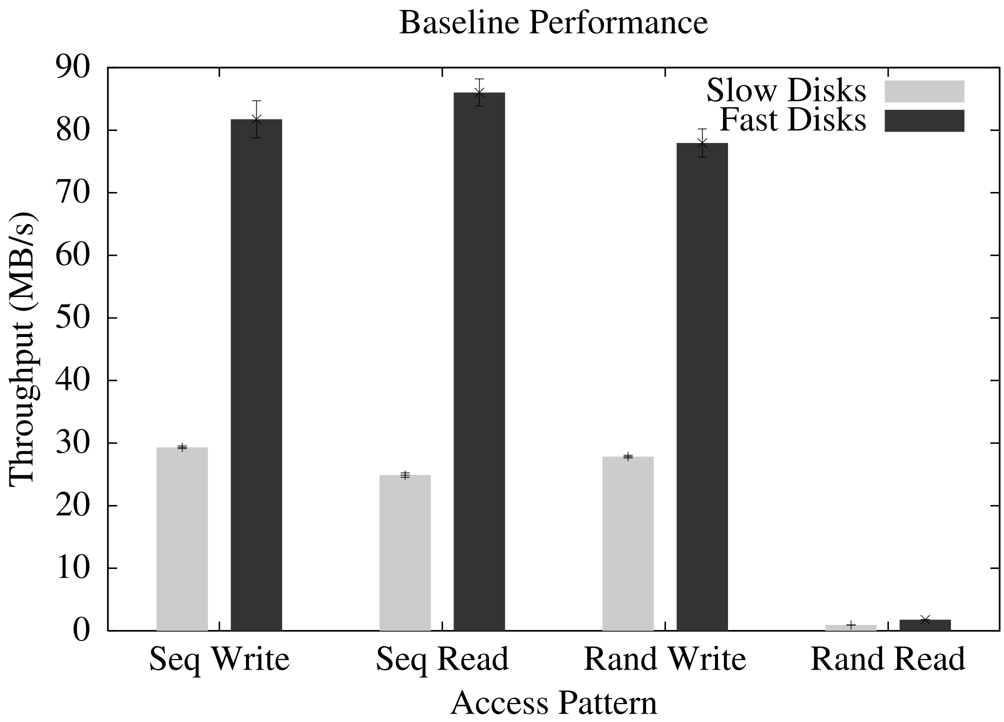


Figure 3.3 **Baseline Performance Comparison.** The figure plots the performance of I·LFS/E×RAID under sequential writes, sequential reads, random writes, and random reads. The tests are run on four disks, varying whether the disks used are the four slow disks or the four fast ones. In all cases, requests generated by the tests are 8 KB in size, and the total data-set size is 200 MB.

As we can see in Figure 3.3, sequential write, sequential read, and random writes all perform excellently, achieving high bandwidth across both disk configurations. Not surprisingly for a log-based file system, random reads perform much more poorly, achieving roughly 0.9 MB/s on the four slow disks, and 1.8 MB/s on the four fast disks, in line with what one would expect from these disks in a typical RAID configuration.

3.4.2 On-line Expansion

We now demonstrate the performance of the system under writes as disks are added to the system on-line. In this experiment, the disks are already present within the PC, and thus the expansion stresses the software infrastructure and not hardware capabilities.

Figure 3.4 plots the performance of sequential writes over time as disks are added to the system (random writes perform similarly, due to the nature of LFS). Along the x-axis, the amount of data written to disk is shown, and the y-axis plots the rate that the most recent 64 MB was committed to disk. As one can see from the graph, I·LFS immediately starts using the disks for write traffic as they are added to the system. However, read traffic will continue to be directed to the original disks for older data. The LFS cleaner could redistribute existing data over the newly-added disks, either explicitly or through cleaning, but we have not yet explored this possibility.

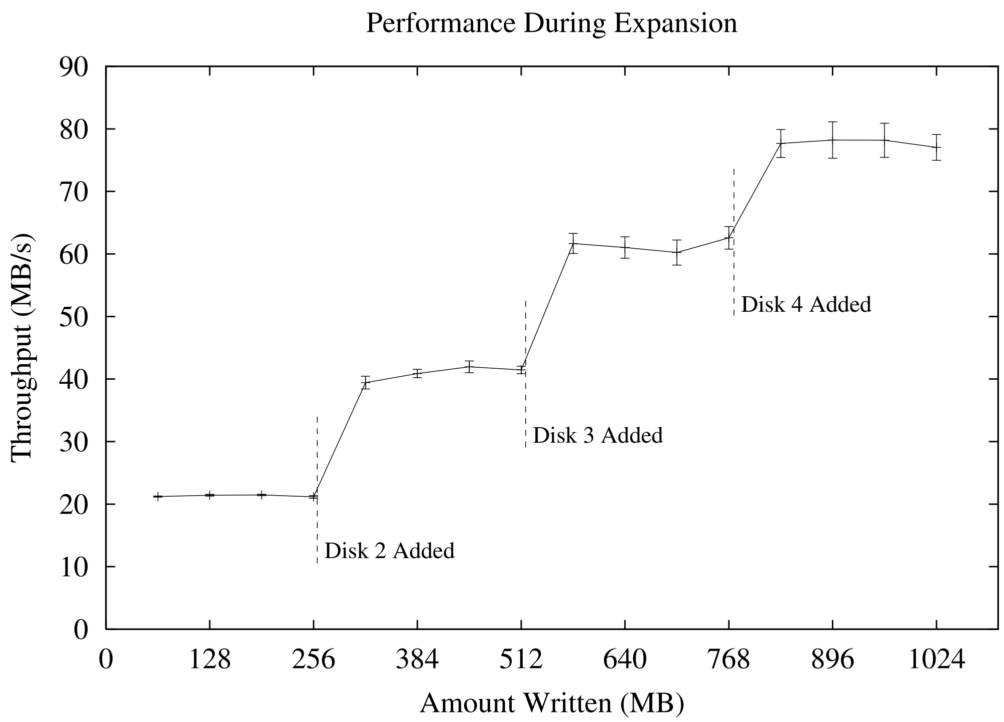


Figure 3.4 **Storage Expansion.** The graph plots the performance of I·LFS during storage expansion. The experiment begins with I·LFS writing to a single disk. Each time 256 MB is written, a new disk is brought on-line, and I·LFS immediately begins writing to it for increased performance. Disk expansion is accomplished via a simple command, which adds the disk (or region) to the file system without down time.

3.4.3 Dynamic Parallelism

We next explore the ability of I·LFS to place segments dynamically in different regions based on the current performance characteristics of the system, in order to demonstrate the ability of I·LFS to react to static and dynamic performance differences across devices.

There are many reasons for performance variation among drives. For example, when new disks are added, they can likely be faster than older ones; further, unexpected dynamic performance variations due to bad-block remapping or "hot spots" in the workload are not uncommon [6], and therefore can also lead to performance heterogeneity across disks. Indeed, the ability to expand the disk system on-line (as shown above) induces a workload imbalance, as read traffic is not directed to the newly-added disks until the cleaner has reorganized data across all of the disks in the system.

We experiment with both static and dynamic performance variations in this subsection. Figure 3.5 shows the results of our static heterogeneity test. The sequential write performance of I·LFS with its dynamic segment placement scheme is plotted along with FFS on top of the NetBSD concatenated disk driver (CCD) configured to stripe data in a RAID-0 fashion. In all experiments, data is written to four disks. Along the x-axis, we increase the number of slow disks in the system; thus, at the extreme left, all of the four disks are fast ones, at the right they are all slow ones, and in the middle are different heterogeneous configurations.

As we can see in the figure, by writing segments dynamically in proportion to delivered disk performance, I·LFS/E×RAID is able to deliver the full bandwidth of the underlying storage system to applications – overall performance degrades gracefully as more slow disks replace fast ones in

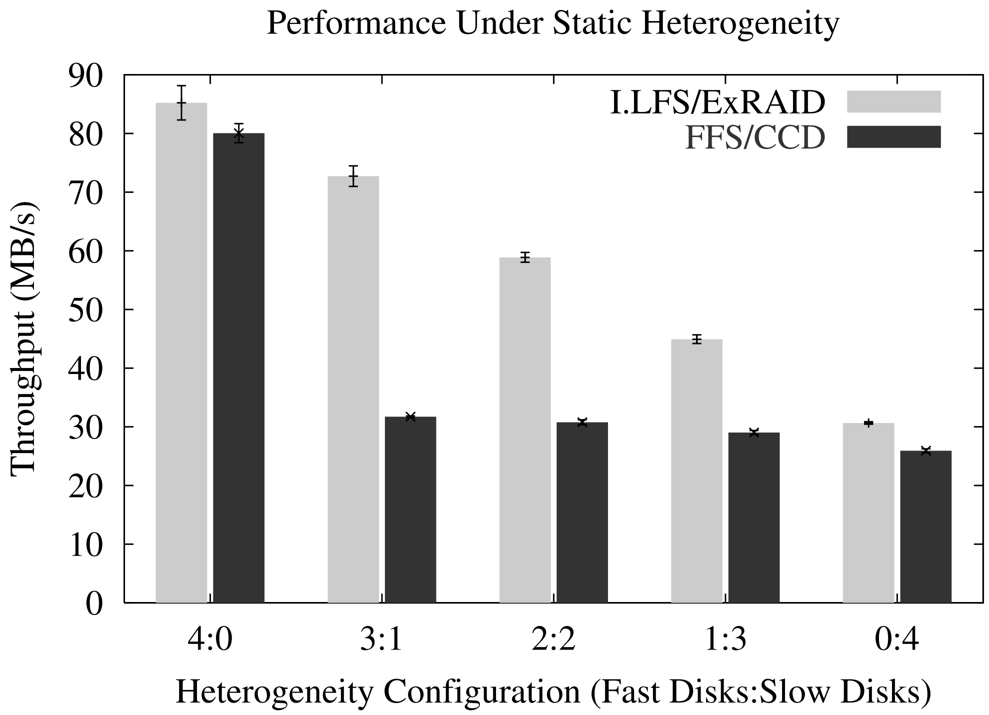


Figure 3.5 **Static Storage Heterogeneity.** The figure plots the performance of I·LFS versus FFS/CCD with standard RAID-0 striping, both under a series of disk configurations. Along the x-axis, the number of fast and slow disks are varied (f:s implies f fast disks and s slow ones). By adjusting where segments are written dynamically, I·LFS/E×RAID is able to deliver the full bandwidth of disks. In contrast, standard striping performs at the rate of the slowest disk in the system. For each test, 200 MB is written to disk.

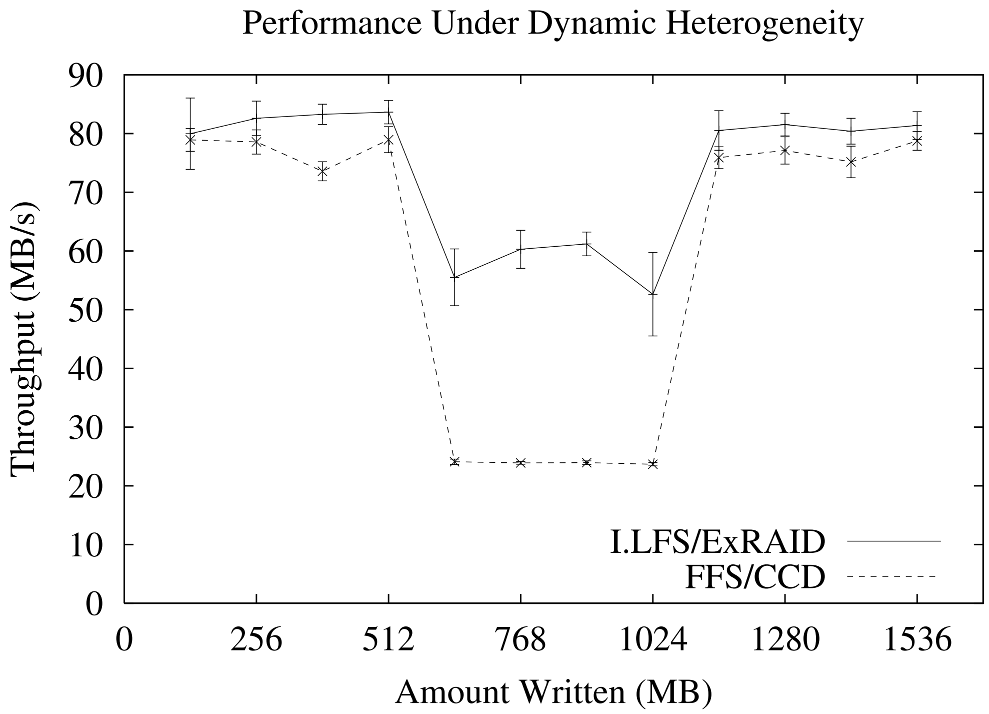


Figure 3.6 **Dynamic Storage Heterogeneity.** The figure plots the performance of I·LFS/E×RAID and FFS/CCD under a dynamic performance variation. During the experiment, the performance of a single disk is temporarily degraded; the faulty disk delays requests for a fixed time, reducing throughput of the disk from 21.6 MB/s to 5.8 MB/s. By adaptively writing more data to the other disks, I·LFS/E×RAID with dynamic segment placement is better able to adjust to the imbalance and deliver higher throughput.

the storage system. RAID-0 striping performs at the rate of the slowest disk, and thus performs poorly in any heterogeneous configuration.

We also perform a "misconfiguration" test. In this experiment, we configure the storage system to utilize two partitions on the *same* disk, emulating a misconfiguration by an administrator (similar in spirit to tests performed by Brown and Patterson [10]). Thus, while the disk system appears to contain four separate disks, it really only contains three. In this case, I·LFS/E×RAID writes data to disk at 65 MB/s, whereas standard striping delivers only 46 MB/s. The dynamic segment striping of I·LFS is successfully able to balance load across the disks, in this case properly assigning less load to each partition within the accidentally over-burdened disk.

In our final heterogeneity experiment, we introduce an artificial "performance fault" into a storage system consisting of four fast disks, in order to confirm that our load balancing works well in the face of dynamic performance variations. Figure 3.6 shows the performance during a write of both I·LFS/E×RAID with dynamic segment placement and FFS/CCD using RAID-0 striping in a case where a single disk of the four exhibits a performance degradation. After one third of the data is written, a kernel-based utility is used to temporarily delay completed requests from one of the disks. The delay has the effect of reducing its throughput from 21.6 MB/s to 5.8 MB/s. The impaired disk is returned to normal operation after an additional one third of the data is written. As we can see from the figure, I·LFS/E×RAID does a better job of tolerating the fluctuations induced during the second phase of the experiment, improving performance by over a factor of two as compared to FFS/CCD.

3.4.4 Flexible Redundancy

In our first redundancy experiment, we verify the operation of our system in the face of failure. Figure 3.7 plots the performance of a set of processes performing random reads from redundant files on I·LFS. Initially, the bandwidth of all four disks is utilized by balancing the read load across the mirrored copies of the data. As the experiment progresses, a disk failure is simulated by disabling reads to one of the disks. I·LFS continues providing data from the available replicas, but overall performance is reduced.

Next, we demonstrate the flexibility of per-file redundancy when the redundancy is managed by the file system. A total of 20 files are written concurrently to a system consisting of four fast disks, while the percentage of those files that are mirrored is increased along the x-axis. The results are shown in Figure 3.8.

As expected, the net throughput of the system decreases linearly as more files are mirrored, and when all are mirrored, overall throughput is roughly halved. Thus, with per-file redundancy, users "get what they pay for"; if users want a file to be redundant, the performance cost of replication is paid during the write, and if not, the performance of the write reflects the full bandwidth of the underlying disks.

3.4.5 Lazy Mirroring

In our final experiment, we demonstrate some of the performance characteristics of lazy mirroring. Figure 3.9 plots the write performance to a set of lazily mirrored files. After a delay of 20 seconds, the cleaner begins replicating data, and the normal file system traffic suffers from a small

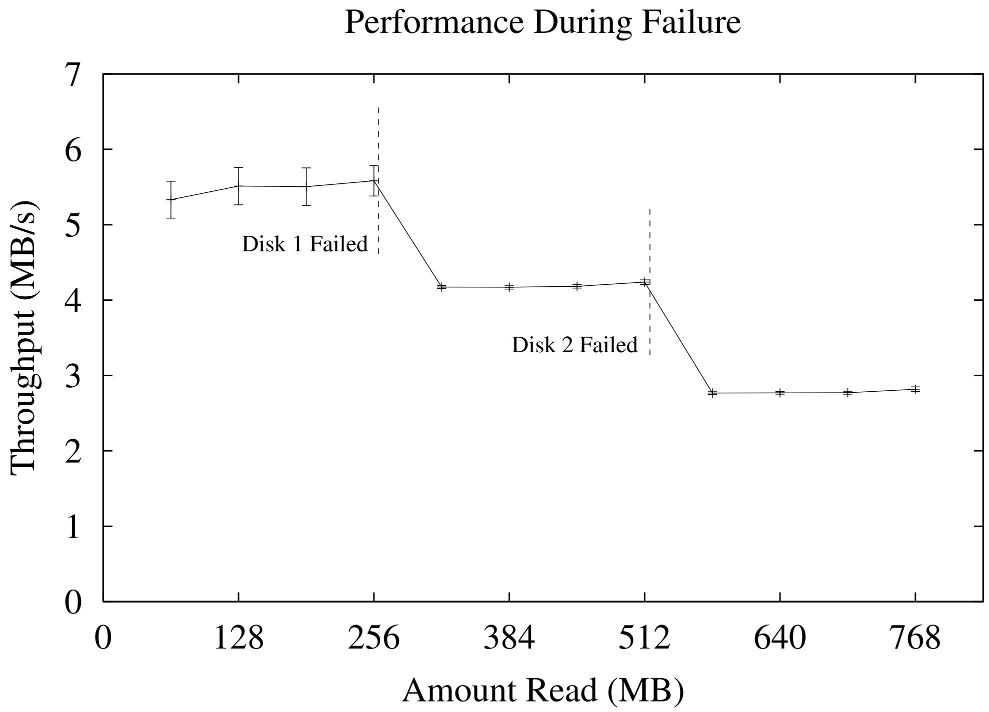


Figure 3.7 **Storage Failure.** The figure plots the random read performance to a set of mirrored files across four disks on I·LFS. At the labeled points in the graph, a disk is taken off-line, and performance decreases because I·LFS can no longer balance the read load between the replicas. Note that in this example, I·LFS/E×RAID can survive any single disk failure; however, after the first failure, I·LFS/E×RAID can only tolerate the loss of the other disk in the set.

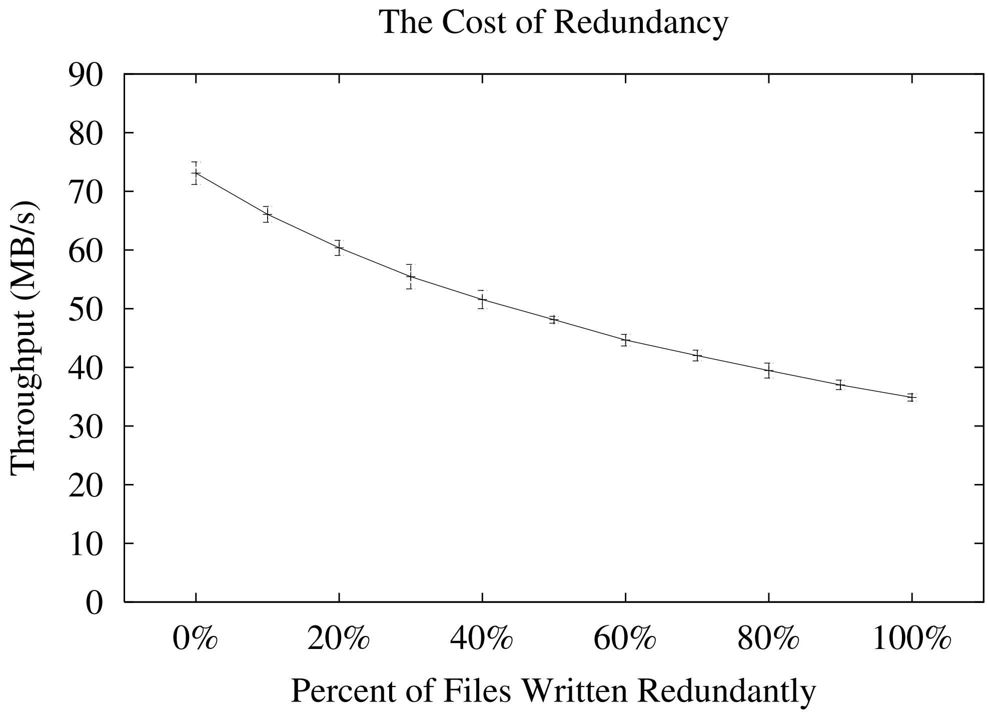


Figure 3.8 **Per-file Redundancy.** The figure plots the performance of writes to 20 separate files as the percent of those files that are mirrored increases. As more files are mirrored, the net bandwidth of the system drops to roughly half of its peak rate, as expected. The peak bandwidth achieved is lower than the previous experiments due to the increased number of files and subsequent meta-data operations. In each experiment, 200 MB is written out to disk.

decline in performance. The default replication delay for the system is two minutes in length, but an abbreviated delay is used here to reduce the time of the experiments.

From the figure, we can see the potential benefits of lazy mirroring, as well as its potential costs. If lazily mirrored files are indeed deleted before replication begins, the full throughput of the storage layer will be realized. However, if many or all lazily mirrored files are not deleted before replication, the system incurs an extra penalty, as those files must be read back from disk and then replicated, which will affect subsequent file system traffic. Therefore, lazy mirroring should be used carefully, either in systems with highly bursty traffic (*i.e.*, idle time for the lazy replicas to be created), or with files that are easily distinguishable as short-lived.

3.5 Discussion

In implementing I·LFS/E×RAID, we were concerned that by pushing more functionality into the file system, the code would become unmanageably complex. Thus, one of our primary goals is to minimize code complexity. We believe we achieve this goal, integrating the three major pieces of functionality with only an additional 1,500 lines of code, a 19% increase over the original size of the LFS implementation. Of this additional code, roughly half is due to the redundancy management.

From the design standpoint, we find that managing redundancy within the file system has many benefits, but also causes many difficulties. For example, to solve the crossed-pointer problem, we applied a divide-and-conquer technique. By placing the primary copy of a file into one of two sets, and its mirror in the other, we enable fast operation under failure. However, our solution limits data placement flexibility, in that once a file is assigned to a set, any subsequent writes to that file must be written to that set. This limitation affects performance, particularly under heterogeneous configurations where one set has significantly different performance characteristics than the other. Though we can relax these placement restrictions, *e.g.*, by choosing which disks constitute a set on a per-file basis, the problem is fundamental to our approach to file-system management of redundancy.

From the implementation standpoint, file-system managed redundancy is also problematic, in that the vnode layer is designed with a single underlying disk in mind. Though our recursive invocation technique was successful, it stretched the limits of what was possible in the current framework, and new additions or modifications to the code are not always straightforward to implement. To truly support file-system managed redundancy, a redesign of the vnode layer may be beneficial [56].

3.6 Conclusions

In terms of abstractions, block-level storage systems such as SCSI have been quite successful: disks hide low-level details from file systems such as the exact mechanics of arm movement and head positioning, but still export a simple performance model upon which file systems could optimize. As Lampson said: "[...] an interface can combine simplicity, flexibility, and high performance together by solving one problem and leaving the rest to the client" [33]. In early single-disk systems, this balance was struck nearly perfectly.

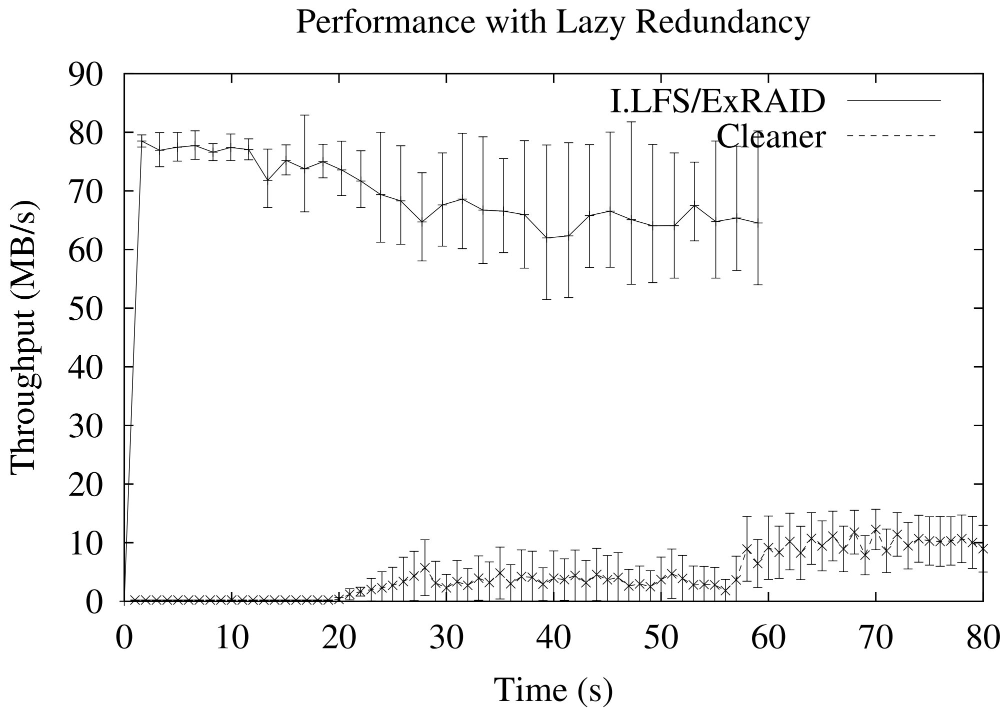


Figure 3.9 **Lazy Mirroring.** The figure plots the write performance to a set of lazy redundant files on I·LFS with a replication delay of 20 seconds. Peak performance is achieved during the initial portion of the test, but performance is reduced slightly as the cleaner begins replicating data. After the write test completes, the cleaner continues to replicate data in the background.

As storage systems evolved from a single drive into a RAID with multiple disks, the interface remained simple, but the RAID itself did not. The result is a system full of misinformation: the file system no longer has an accurate model of disk behavior, and the now-complex storage system does not have a good understanding of what to expect from the file system.

E$\times$RAID and I·LFS bridge this information gap by design: the presence of multiple regions is exposed directly to the file system, enabling new functionality. In this chapter, we have explored the implementation of on-line expansion, dynamic parallelism, flexible redundancy, and lazy mirroring in I·LFS. All were implemented in a relatively straight-forward manner within the file system, increasing system manageability, performance, and functionality, while maintaining a reasonable level of overall system complexity. Some of these aspects of I·LFS would be difficult to build in the traditional storage stack, highlighting the importance of informing interfaces that allow functionality to be placed in the correct layer of the system.

Chapter 4

Collaborating Layers: Journal-guided Resynchronization

4.1 Introduction

In this chapter, we look beyond information-only interfaces to new interfaces that allow storage stack layers to communicate more effectively. Our goal is to identify a problem that is best solved in a coordinated manner across layers, and to develop an efficient, informing interface that allows for such collaboration. Specifically, we address the problem of maintaining consistency at the RAID level.

The task of a RAID is to maintain consistency between the data and the redundant information it stores. This invariant provides the ability to recover data in the case of a disk failure. However, because the blocks reside on more than one disk, updates cannot be applied atomically. Hence, maintaining these invariants in the face of crashes is challenging. If a crash occurs during a write to an array, its blocks may be left in an inconsistent state. This inconsistency introduces a *window of vulnerability*; if a data disk fails before the stripe is made consistent, the data on that disk will be lost. Automatic reconstruction of the missing data block, based on the inconsistent parity, will silently return bad data to the client.

High-end storage arrays circumvent this problem gracefully with non-volatile memory. By buffering an update in NVRAM until the disks have been consistently updated, a hardware-based approach avoids the window of vulnerability entirely. The outcome is ideal: both performance and reliability are excellent. Unfortunately, the extra hardware entails extra cost; many of these solutions come with multi-million dollar price tags [21].

In commodity RAID systems that lack non-volatile memory, a performance versus reliability trade-off must be made. Most current software RAID implementations choose performance over reliability [78]: they simply issue writes to the disks in parallel, hoping that an untimely crash does not occur in between. If a crash does occur, these systems employ an expensive *resynchronization* process: by scanning the entire volume, such discrepancies can be found and repaired. For large volumes, this process can take hours or even days.

The alternate software RAID approach chooses reliability over performance [16]. By applying write-ahead logging within the array to record the location of pending updates before they are issued, these systems avoid time-consuming resynchronization: during recovery, the RAID simply repairs the locations as recorded in its log. Unfortunately, removing the window of vulnerability comes with a high performance cost: each update within the RAID must now be preceded by a synchronous write to the log, greatly increasing the total I/O load on the disks.

To solve the consistent update problem in the commodity RAID environment, and to develop a solution with both high performance and reliability, we take a global view of the storage stack: can we find a *collaborative* approach that leverages functionality in other layers of the system to assist us? In many cases, the client of the RAID system will be a modern journaling file system, such as the default Linux file system, ext3 [80, 81, 82], or ReiserFS [51], JFS [8], or Windows NTFS [71]. Although standard journaling techniques maintain the consistency of file system data structures, they do not solve the consistent update problem at the RAID level. We find, however, that journaling can be readily augmented to do so.

Specifically, we introduce a new mode of operation within Linux ext3: *declared mode*. Before writing to any permanent locations, declared mode records its intentions in the file system journal. This functionality guarantees a record of all outstanding writes in the event of a crash. By consulting this activity record, the file system knows which blocks were in the midst of being updated and hence can dramatically reduce the window of vulnerability following a crash.

To complete the process, the file system must be able to communicate its information about possible vulnerabilities to the RAID layer below. For this purpose, we add a new informing interface to the RAID layer: the *verify read*. Upon receiving a verify read request, the RAID layer reads the requested block as well as its mirror or parity group and verifies the redundant information. If an irregularity is found, the RAID layer re-writes the mirror or parity to produce a consistent state.

We combine these features to integrate journal-guided resynchronization into the file system recovery process. Using our record of write activity vastly decreases the time needed for resynchronization, in some cases from a period of days to mere seconds. Hence, our approach avoids the performance versus reliability trade-off found in commodity RAID systems: performance remains high and the window of vulnerability is greatly reduced.

In general, we believe the key to our solution is its *collaborative* nature. By removing the strict isolation between the file system above and the RAID layer below, these two subsystems can work *together* to solve the consistent update problem without sacrificing either performance or reliability.

The rest of the chapter is organized as follows. Section 4.2 illustrates the RAID consistent update problem and quantifies the likelihood that a crash will lead to data vulnerability. Section 4.3 provides an introduction to the ext3 file system and its operation. In Section 4.4, we analyze ext3's write activity, introduce ext3 declared mode and an addition to the RAID interface, and merge RAID resynchronization into the journal recovery process. Section 4.5 evaluates the performance of declared mode and the effectiveness of journal-guided resynchronization. We conclude in Section 4.6.

4.2 The Consistent Update Problem

4.2.1 Introduction

The task of a RAID is to maintain an invariant between the data and the redundant information it stores. These invariants provide the ability to recover data in the case of a disk failure. For RAID-1, this means that each mirrored block contains the same data. For parity schemes, such as RAID-5, this means that the parity block for each stripe stores the exclusive-or of its associated data blocks.

However, because the blocks reside on more than one disk, updates cannot be applied atomically. Hence, maintaining these invariants in the face of failure is challenging. If a crash occurs during a write to an array, its blocks may be left in an inconsistent state. Perhaps only one mirror was successfully written to disk, or a data block may have been written without its parity update.

We note here that the consistent update problem and its solutions are distinct from the traditional problem of RAID disk failures. When such a failure occurs, all of the redundant information in the array is lost, and thus all of the data is vulnerable to a second disk failure. This situation is solved by the process of reconstruction, which regenerates all of the data located on the failed disk.

4.2.2 Failure Models

We illustrate the consistent update problem with the example shown in Figure 4.1. The diagram depicts the state of a single stripe of blocks from a four disk RAID-5 array as time progresses from left to right. The software RAID layer residing on the machine is servicing a write to data block **Z**, and it must also update the parity block, **P**. The machine issues the data block write at time 1, it is written to disk at time 3, and the machine is notified of its completion at time 4. Similarly, the parity block is issued at time 2, written at time 5, and its notification arrives at time 6. After the data write to block **Z** at time 3, the stripe enters a *window of vulnerability*, denoted by the shaded blocks. During this time, the failure of any of the first three disks will result in data loss. Because the stripe's data and parity blocks exist in an inconsistent state, the data residing on a failed disk cannot be reconstructed. This inconsistency is corrected at time 5 by the write to **P**.

We consider two failure models to allow for the possibility of independent failures between the host machine and the array of disks. We will discuss each in turn and relate their consequences to the example in Figure 4.1. The *machine failure model* includes events such as operating system crashes and machine power losses. In our example, if the machine crashes between times 1 and 2,

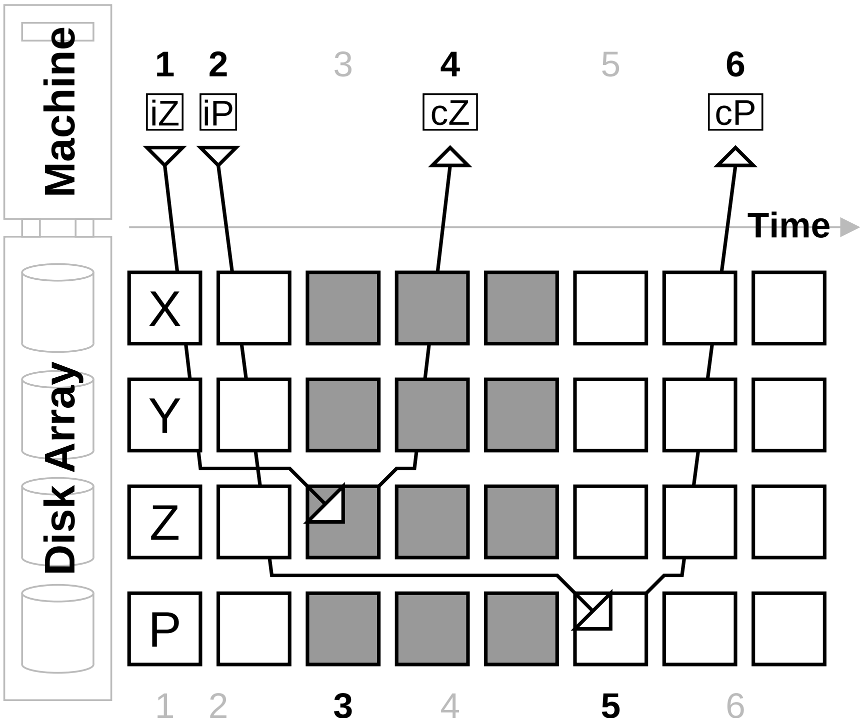


Figure 4.1 **Failure Scenarios.** The diagram illustrates the sequence of events for a data block write and a parity update to a four disk RAID-5 array as time progresses from left to right. The boxes labeled *i* indicate a request being issued, and those labeled *c* represent completions. The shaded blocks denote a window of vulnerability.

and the array remains active, the stripe will be left in an inconsistent state after the write completes at time 3.

Our second model, the *disk failure model*, considers power losses at the disk array. If such a failure occurs between time 3 and time 5 in our example, the stripe will be left in a vulnerable state. Note that the disk failure model encompasses non-independent failures such as a simultaneous power loss to the machine and the disks.

4.2.3 Measuring Vulnerability

To determine how often a crash or failure could leave an array in an inconsistent state, we instrument the Linux software RAID-5 layer and the SCSI driver to track several statistics. First, we record the amount of time between the first write issued for a stripe and the last write issued for a stripe. This measures the difference between times 1 and 2 in Figure 4.1, and corresponds directly to the period of vulnerability under the machine failure model.

Second, we record the amount of time between the first write completion for a stripe and the last write completion for a stripe. This measures the difference between time 4 and time 6 in our example. Note, however, that the vulnerability under the disk failure model occurs between time 3 and time 5, so our measurement is an approximation. Our results may slightly overestimate or underestimate the actual vulnerability depending on the time it takes each completion to be sent to and processed by the host machine. Finally, we track the number of stripes that are vulnerable for each of the models. This allows us to calculate the percent of time that any stripe in the array is vulnerable to either type of failure.

Our test workload consists of multiple threads performing synchronous, random writes to a set of files on the array. All of the experiments in this chapter are performed on an Intel Pentium Xeon 2.6 GHz processor with 512 MB of RAM running Linux kernel 2.6.11. The machine has five IBM 9LZX disks configured as a 1 GB software RAID-5 array. The RAID volume is sufficiently large to perform our benchmarks yet small enough to reduce the execution time of our resynchronization experiments.

Figure 4.2 plots the percent of time (over the duration of the experiment) that any array stripe is vulnerable as the number of writers in the workload is increased along the x-axis. As expected, the cumulative window of vulnerability increases as the amount of concurrency in the workload is increased. The vulnerability under the disk failure model is greater because it is dependent on the response time of the write requests. Even for a small number of writers, it is more than likely that a disk failure will result in an inconsistent state. For higher concurrency, the array exists in a vulnerable state for up to 80% of the length of the experiment.

The period of vulnerability under the machine failure model is lower because it depends only on the processing time needed to issue the write requests. In our experiment, vulnerability reaches approximately 40%. At much higher concurrencies, however, the ability to issue requests could be impeded by full disk queues. In this case, the machine vulnerability will also depend on the disk response time and will increase accordingly.

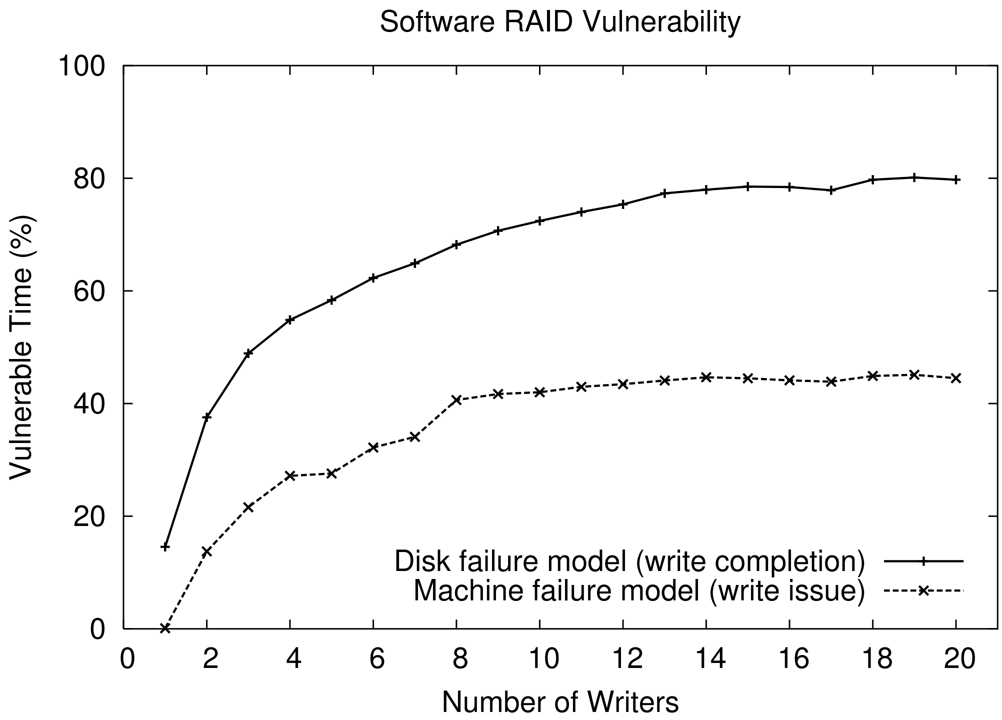


Figure 4.2 **Software RAID Vulnerability.** The graph plots the percent of time (over the duration of the experiment) that an inconsistent disk state exists in the RAID-5 array as the number of writers increases along the x-axis. Vulnerabilities due to disk failure and machine failure are plotted separately.

4.2.4 Solutions

To solve this problem, high-end RAID systems make use of non-volatile storage, such as NVRAM. When a write request is received, a log of the request and the data are first written to NVRAM, and then the updates are propagated to the disks. In the event of a crash, the log records and data present in the NVRAM can be used to replay the writes to disk, thus ensuring a consistent state across the array. This functionality comes at an expense, not only in terms of raw hardware, but in the cost of developing and testing a more complex system.

Software RAID, on the other hand, is frequently employed in commodity systems that lack non-volatile storage. When such a system reboots from a crash, there is no record of write activity in the array, and therefore no indication of where RAID inconsistencies may exist. Linux software RAID rectifies this situation by laboriously reading the contents of the entire array, checking the redundant information, and correcting any discrepancies. For RAID-1, this means reading both data mirrors, comparing their contents, and updating one if their states differ. Under a RAID-5 scheme, each stripe of data must be read and its parity calculated, checked against the parity on disk, and re-written if it is incorrect.

This approach fundamentally affects both reliability and availability. The time-consuming process of scanning the entire array lengthens the window of vulnerability during which inconsistent redundancy may lead to data loss under a disk failure. Additionally, the disk bandwidth devoted to resynchronization has a deleterious effect on the foreground traffic serviced by the array. Consequently, there exists a fundamental tension between the demands of reliability and availability:

allocating more bandwidth to recover inconsistent disk state reduces the availability of foreground services, but giving preference to foreground requests increases the time to resynchronize.

As observed by Brown and Patterson [10], the default Linux policy addresses this trade-off by favoring availability over reliability, limiting resynchronization bandwidth to 1000 KB/s per disk. Unfortunately, such a slow rate may equate to days of repair time and vulnerability for even moderately sized arrays of hundreds of gigabytes. Figure 4.3 illustrates this problem by plotting an analytical model of the resynchronization time for a five disk array as the raw size of the array increases along the x-axis. With five disks, the default Linux policy will take almost four minutes of time to scan and repair each gigabyte of disk space, which equates to *two and a half days* for a terabyte of capacity. Disregarding the availability of the array, even modern interconnects would need approximately an hour at their full bandwidth to resynchronize the same one terabyte array.

One possible solution to this problem is to add logging to the RAID system in a manner similar to that discussed above. This approach suffers from two drawbacks, however. First, logging to the array disks themselves would likely decrease the overall performance of the array by interfering with foreground requests. The high-end solution discussed previously benefits from fast, independent storage in the form of NVRAM. Second, adding logging and maintaining an acceptable level of performance could add considerable complexity to the software. For instance, the Linux software RAID implementation uses little buffering, discarding stripes when their operations are complete. A logging solution, however, may need to buffer requests significantly in order to batch updates to the log and improve performance.

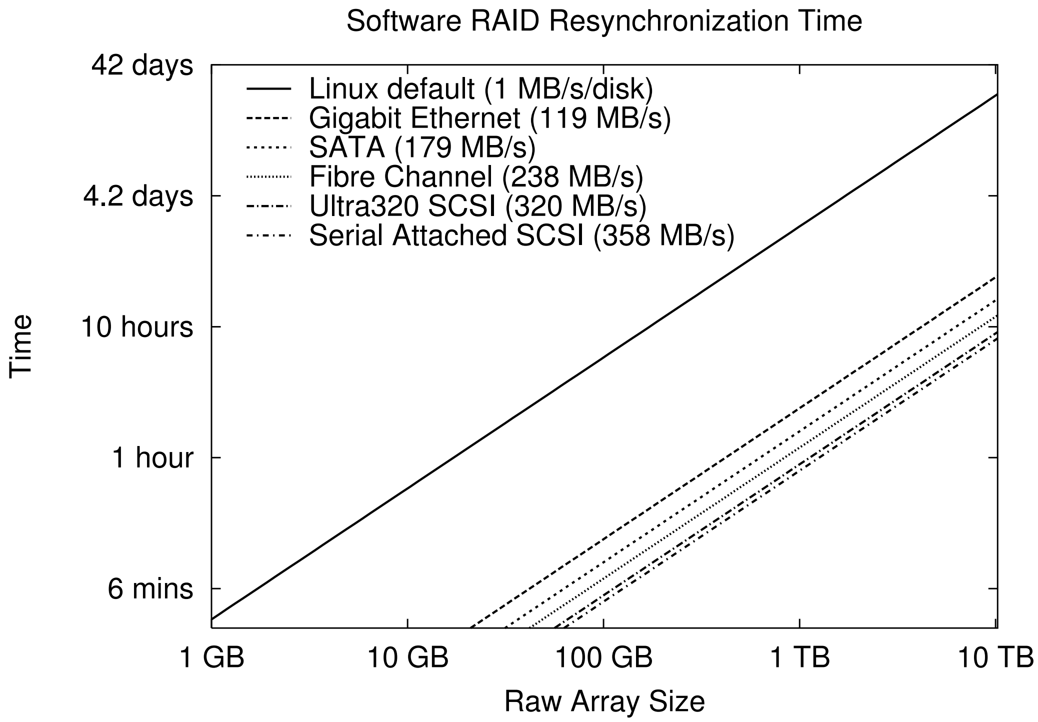


Figure 4.3 **Software RAID Resynchronization Time.** The graph plots the time to resynchronize a five disk array as the raw capacity increases along the x-axis.

Another solution is to perform intent logging to a bitmap representing regions of the array. This mechanism is used by the Solaris Volume Manager [74] and the Veritas Volume Manager [86] to provide optimized resynchronization. An implementation for Linux software RAID is also in development [16], though it has not been merged into the main kernel. Like logging to the array, this approach is likely to suffer from poor performance. For instance, the Linux implementation performs a synchronous write to the bitmap before updating data in the array to ensure proper resynchronization. Performance may be improved by increasing the bitmap granularity, but this comes at the cost of performing scan-based resynchronization over larger regions.

Software RAID is just one layer in the storage hierarchy. One likely configuration contains a modern journaling file system in the layer above, logging disk updates to maintain consistency across its on-disk data structures. In the next sections, we examine how a journaling file system can be used to solve the RAID resynchronization problem.

4.3 ext3 Background

In this section, we discuss the Linux ext3 file system, its operation, and its data structures. These details will be useful in our analysis of its write activity and the description of our modifications to support journal-guided resynchronization in Section 4.4. Although we focus on ext3, we believe our techniques are general enough to apply to other journaling file systems, such as ReiserFS and JFS for Linux, and NTFS for Windows.

Linux ext3 is a modern journaling file system that aims to keep complex on-disk data structures in a consistent state. To do so, all file system updates are first written to a log called the journal.

Once the journal records are stored safely on disk, the updates can be applied to their home locations in the main portion of the file system. After the updates are propagated, the journal records are erased and the space they occupied can be re-used.

This mechanism greatly improves the efficiency of crash recovery. After a crash, the journal is scanned and outstanding updates are replayed to bring the file system into a consistent state. This approach constitutes a vast improvement over the previous process (*i.e.* fsck [41]) that relied on a full scan of the file system data structures to ensure consistency. It seems natural, then, to make use of the same journaling mechanism to improve the process of RAID resynchronization after a crash.

4.3.1 Modes

The ext3 file system offers three modes of operation: data-journaling mode, ordered mode, and writeback mode. In data-journaling mode, all data and metadata is written to the journal, coordinating all updates to the file system. This provides very strong consistency semantics, but at the highest cost. All data written to the file system is written twice: first to the journal, then to its home location.

Ordered mode, the ext3 default, writes all file system metadata to the journal, but file data is written directly to its home location. In addition, this mode guarantees a strict ordering between the writes: all file data for a transaction is written to disk before the corresponding metadata is written to the journal and committed. This guarantees that file metadata will never reference a data block before it has been written. Thus, this mechanism provides strong consistency without the expense of multiple writes for file data.

In writeback mode, only file system metadata is written to the journal. Like ordered mode, file data is written directly to its home location; unlike ordered mode, however, writeback mode provides no ordering guarantees between metadata and data, therefore offering much weaker consistency. For instance, the metadata for a file creation may be committed to the journal before the file data is written. In the event of a crash, journal recovery will restore the file metadata, but its contents could be filled with arbitrary data. We will not consider writeback mode for our purposes because of its weaker consistency and its lack of write ordering.

4.3.2 Transaction Details

To reduce the overhead of file system updates, sets of changes are grouped together into compound transactions. These transactions exist in several phases over their lifetimes. Transactions start in the *running* state. All file system data and metadata updates are associated with the current running transaction, and the buffers involved in the changes are linked to the in-memory transaction data structure. In ordered mode, data associated with the running transaction may be written at any time by the kernel pdflush daemon, which is responsible for cleaning dirty buffers. Periodically, the running transaction is closed and a new transaction is started. This may occur due to a timeout, a synchronization request, or because the transaction has reached a maximum size.

Next, the closed transaction enters the *commit* phase. All of its associated buffers are written to disk, either to their home locations or to the journal. After all of the transaction records reside safely in the journal, the transaction moves to the *checkpoint* phase, and its data and metadata are copied from the journal to their permanent home locations. If a crash occurs before or during the checkpoint of a committed transaction, it will be checkpointed again during the journal *recovery* phase of mounting the file system. When the checkpoint phase completes, the transaction is removed from the journal and its space is reclaimed.

4.3.3 Journal Structure

Tracking the contents of the journal requires several new file system structures. A journal superblock stores the size of the journal file, pointers to the head and tail of the journal, and the sequence number of the next expected transaction. Within the journal, each transaction begins with a *descriptor* block that lists the permanent block addresses for each of the subsequent data or metadata blocks. More than one descriptor block may be needed depending on the number of blocks involved in a transaction. Finally, a *commit* block signifies the end of a particular transaction. Both descriptor blocks and commit blocks begin with a magic header and a sequence number to identify their associated transaction.

4.4 Design and Implementation

The goal of resynchronization is to correct any RAID inconsistencies that result from system crash or failure. If we can identify the outstanding write requests at the time of the crash, we can significantly narrow the range of blocks that must be inspected. This will result in faster resynchronization and improved reliability and availability. Our hope is to recover such a record of outstanding writes from the file system journal. To this end, we begin by examining the write activity generated by each phase of an ext3 transaction.

4.4.1 ext3 Write Analysis

In this section, we examine each of the ext3 transaction operations in detail. We emphasize the write requests generated in each phase, and we characterize the possible disk states resulting from a crash. Specifically, we classify each write request as targeting a known location, an unknown location, or a bounded location, based on its record of activity in the journal. Our goal, upon restarting from a system failure, is to recover a record of *all outstanding write requests* at the time of the crash.

Running:

1. In ext3 ordered mode, the pdflush daemon may write dirty pages to disk while the transaction is in the running state. If a crash occurs in this state, the affected locations will be unknown, as *no record of the ongoing writes will exist in the journal.*

Commit:

1. ext3 writes all un-journaled dirty data blocks associated with the transaction to their home locations, and waits for the I/O to complete. This step applies only to ordered mode, since all data in data-journaling mode is destined for the journal. If a crash occurs during this phase, the locations of any outstanding writes will be unknown.

2. ext3 writes descriptors, journaled data, and metadata blocks to the journal, and waits for the writes to complete. In ordered mode, only metadata blocks will be written to the journal, whereas all blocks are written to the journal in data-journaling mode. If the system fails during this phase, no specific record of the ongoing writes will exist, but all of the writes will be bounded within the fixed location journal.

3. ext3 writes the transaction commit block to the journal, and waits for its completion. In the event of a crash, the outstanding write is again bounded within the journal.

Checkpoint:

1. ext3 writes journaled blocks to their home locations and waits for the I/O to complete. If the system crashes during this phase, the ongoing writes can be determined from the descriptor blocks in the journal, and hence they affect known locations.

2. ext3 updates the journal tail pointer in the superblock to signify completion of the checkpointed transaction. A crash during this operation involves an outstanding write to the journal superblock, which resides in a known, fixed location.

Recovery:

1. ext3 scans the journal checking for the expected transaction sequence numbers (based on the sequence in the journal superblock) and records the last committed transaction.

2. ext3 checkpoints each of the committed transactions in the journal, following the steps specified above. All write activity occurs to known locations.

Block Type	Data-journaling Mode
superblock	known, fixed location
journal	bounded, fixed location
home metadata	known, journal descriptors
home data	known, journal descriptors

Block Type	Ordered Mode
superblock	known, fixed location
journal	bounded, fixed location
home metadata	known, journal descriptors
home data	unknown

Table 4.1 **Journal Write Records.** The table lists the block types written during transaction processing and how their locations can be determined after a crash.

Table 4.1 summarizes our ability to locate ongoing writes after a crash for the data-journaling and ordered modes of ext3. In the case of data-journaling mode, the locations of any outstanding writes can be determined (or at least bounded) during crash recovery, be it from the journal descriptor blocks or from the fixed location of the journal file and superblock. Thus, the existing ext3 data-journaling mode is quite amenable to assisting with the problem of RAID resynchronization. On the down side, however, data-journaling typically provides the least performance of the ext3 family.

For ext3 ordered mode, on the other hand, data writes to permanent home locations are not recorded in the journal data structures, and therefore cannot be located during crash recovery. We now address this deficiency with a modified ext3 ordered mode: declared mode.

4.4.2 ext3 Declared Mode

In the previous section we concluded that, if a crash occurs while writing data directly to its permanent location, the ext3 ordered mode journal will contain no record of those outstanding writes. The locations of any RAID level inconsistencies caused by those writes will remain unknown upon restart. To overcome this deficiency, we introduce a new variant of ordered mode, *declared mode*.

Declared mode differs from ordered mode in one key way: it guarantees that a write record for each data block resides safely in the journal before that location is modified. Effectively, the file system must *declare its intent* to write to any permanent location before issuing the write.

To keep track of these intentions, we introduce a new journal block, the *declare* block. A set of declare blocks is written to the journal at the beginning of each transaction commit phase. Collectively, they contain a list of all permanent locations to which data blocks in the transaction will be written. Though their construction is similar to that of descriptor blocks, their purpose is quite different. Descriptor blocks list the permanent locations for blocks that appear in the journal, whereas declare blocks list the locations of blocks that *do not appear* in the journal. Like descriptor and commit blocks, declare blocks begin with a magic header and a transaction sequence number. Declared mode thus adds a single step to the beginning of the commit phase, which proceeds as follows:

Declared Commit:

1. ext3 writes declare blocks to the journal listing each of the permanent data locations to be written as part of the transaction, and it waits for their completion.

2. ext3 writes all un-journaled data blocks associated with the transaction to their home locations, and waits for the I/O to complete.

3. ext3 writes descriptors and metadata blocks to the journal, and waits for the writes to complete.

4. ext3 writes the transaction commit block to the journal, and waits for its completion.

The declare blocks at the beginning of each transaction introduce an additional space cost in the journal. This cost varies with the number of data blocks each transaction contains. In the best case, one declare block will be added for every 506 data blocks, for a space overhead of 0.2%. In the worst case, however, one declare block will be needed for a transaction containing only a single data block. We investigate the performance consequences of these overheads in Section 4.5.

Implementing declared mode in Linux requires two main changes. First, we must guarantee that no data buffers are written to disk before they have been declared in the journal. To accomplish this, we refrain from setting the dirty bit on modified pages managed by the file system. This prevents the pdflush daemon from eagerly writing the buffers to disk during the running state. The same mechanism is used for all metadata buffers and for data buffers in data-journaling mode, ensuring that they are not written before they are written to the journal.

Second, we need to track data buffers that require declarations, and write their necessary declare blocks at the beginning of each transaction. We start by adding a new *declare tree* to the in-memory transaction structure, and ensure that all declared mode data buffers are placed on this tree instead of the existing *data list*. At the beginning of the commit phase, we construct a set of declare blocks for all of the buffers on the declare tree and write them to the journal. After the writes complete, we simply move all of the buffers from the declare tree to the existing transaction data list. The use of a tree ensures that the writes occur in a more efficient order, sorted by block address. From this point, the commit phase can continue without modification. This implementation minimizes the changes to the shared commit procedure; the other ext3 modes simply bypass the empty declare tree.

4.4.3 Software RAID Interface

Initiating resynchronization at the file system level requires a mechanism to repair suspected inconsistencies after a crash. A viable option for RAID-1 arrays is for the file system to read and re-write any blocks it has deemed vulnerable. In the case of inconsistent mirrors, either the newly written data or the old data will be restored to each block. This achieves the same results as the current RAID-1 resynchronization process. Because the RAID-1 layer imposes no ordering on mirrored updates, it cannot differentiate new data from old data, and merely chooses one block copy to restore consistency.

This read and re-write strategy is unsuitable for RAID-5, however. When the file system re-writes a single block, our desired behavior is for the RAID layer to calculate its parity across the entire stripe of data. Instead, the RAID layer could perform a read-modify-write by reading the target block and its parity, re-calculating the parity, and writing both blocks to disk. This operation depends on the consistency of the data and parity blocks it reads from disk. If they are not consistent, it will produce incorrect results, simply prolonging the discrepancy. In general, then, a new interface is required for the file system to communicate possible inconsistencies to the RAID layer.

We consider two options for the new interface. The first requires the file system to read each vulnerable block and then re-write it with an explicit *reconstruct write* request. In this option, the RAID layer is responsible for reading the remainder of the block's parity group, re-calculating its parity, and then writing the block and the new parity to disk. We are dissuaded from this option because it may perform unnecessary writes to consistent stripes that could cause further vulnerabilities in the event of another crash.

Instead, we opt to add an explicit *verify read* request to the RAID interface. In this case, the RAID layer reads the requested block along with the rest of its stripe and checks to make sure that the parity is consistent. If it is not, the newly calculated parity is written to disk to correct the problem.

The Linux implementation for the verify read request is rather straight-forward. When the file system wishes to perform a verify read request, it marks the corresponding buffer head with a new *RAID synchronize* flag. Upon receiving the request, the software RAID-5 layer identifies the flag and enables an existing *synchronizing* bit for the corresponding stripe. This bit is used to perform the existing resynchronization process. Its presence causes a read of the entire stripe followed by a parity check, exactly the functionality required by the verify read request.

Finally, an option is added to the software RAID-5 layer to disable resynchronization after a crash. This is our most significant modification to the strict layering of the storage stack. The RAID module is asked to entrust its functionality to another component for the overall good of the system. Instead, an apprehensive RAID implementation may delay its own efforts in hopes of receiving the necessary verify read requests from the file system above. If no such requests arrive, it could start its own resynchronization to ensure the integrity of its data and parity blocks.

4.4.4 Recovery and Resynchronization

Using ext3 in either data-journaling mode or declared mode guarantees an accurate view of all outstanding write requests at the time of a crash. Upon restart, we utilize this information and our verify read interface to perform fast, file system guided resynchronization for the RAID layer. Because we make use of the file system journal, and because of ordering constraints between their operations, we combine this process with journal recovery. The dual process of file system recovery and RAID resynchronization proceeds as follows:

Recovery and Resync:

1. ext3 performs verify reads for its superblock and the journal superblock, ensuring their consistency in case they were being written during the crash.

2. ext3 scans the journal checking for the expected transaction sequence numbers (based on the sequence in the journal superblock) and records the last committed transaction.

3. For the first committed transaction in the journal, ext3 performs verify reads for the home locations listed in its descriptor blocks. This ensures the integrity of any blocks undergoing checkpoint writes at the time of the crash. Only the first transaction need be examined because checkpoints must occur in order, and each checkpointed transaction is removed from the journal before the next is processed. Note that these verify reads must take place before the writes are replayed below to guarantee the parity is up-to-date. Adding the explicit reconstruct write interface mentioned earlier would negate the need for this two step process.

4. ext3 issues verify reads beyond the last committed transaction (at the head of the journal) for the length of the maximum transaction size. This corrects any inconsistent blocks as a result of writing the next transaction to the journal.

5. While reading ahead in the journal, ext3 identifies any declare blocks and descriptor blocks for the next uncommitted transaction. If no descriptor blocks are found, it performs verify reads for the permanent addresses listed in each declare block, correcting any data writes that were outstanding at the time of the crash. Declare blocks from transactions containing descriptors can be ignored, as their presence constitutes evidence for the completion of all data writes to permanent locations.

6. ext3 checkpoints each of the committed transactions in the journal as described in Section 4.4.1.

The implementation re-uses much of the existing framework for the journal recovery process. Issuing the necessary verify reads means simply adding the RAID synchronize flag to the buffers already used for reading the journal or replaying blocks. The verify reads for locations listed in descriptor blocks are handled as the replay writes are processed. The journal verify reads and declare block processing for an uncommitted transaction are performed after the final pass of the journal recovery.

4.5 Evaluation

In this section, we evaluate the performance of ext3 declared mode and compare it to ordered mode and data-journaling mode. We hope that declared mode adds little overhead despite writing extra declare blocks for each transaction. After our performance evaluation, we examine the effects of journal-guided resynchronization. We expect that it will greatly reduce resync time and increase available bandwidth for foreground applications. Finally, we examine the complexity of our implementation.

4.5.1 ext3 Declared Mode

We begin our performance evaluation of ext3 declared mode with two microbenchmarks, random write and sequential write. First, we test the performance of random writes to an existing 100 MB file. A call to fsync() is used at the end of the experiment to ensure that all data reaches disk. Figure 4.4 plots the bandwidth achieved by each ext3 mode as the amount written is increased along the x-axis. All of our graphs plot the mean of five experimental trials.

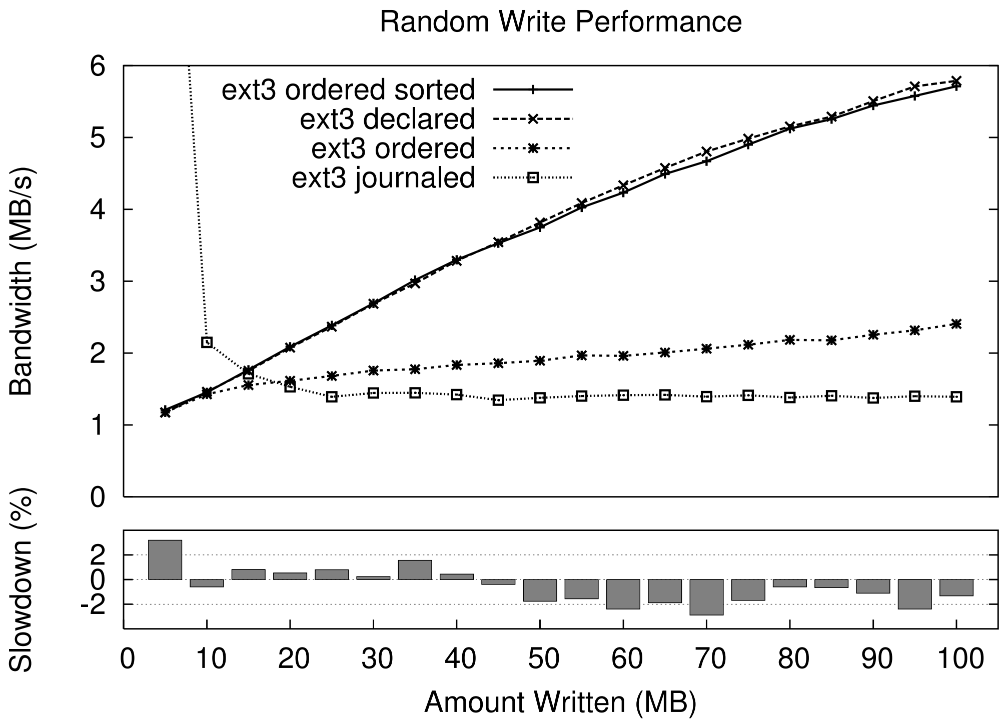


Figure 4.4 **Random Write Performance.** The top graph plots random write performance as the amount of data written is increased along the x-axis. Data-journaling mode achieves 11.07 MB/s when writing 5 MB of data. The bottom graph shows the relative performance of declared mode as compared to ordered mode with sorting.

We identify two points of interest on the graph. First, data-journaling mode underperforms ordered mode as the amount written increases. Note that data-journaling mode achieves 11.07 MB/s when writing only 5 MB of data because the random write stream is transformed into a large sequential write that fits within the journal. As the amount of data written increases, it outgrows the size of the journal. Consequently, the performance of data-journaling decreases because each block is written twice, first to the journal, and then to its home location. Ordered mode garners better performance by writing data directly to its permanent location.

Second, we find that declared mode greatly outperforms ordered mode as the amount written increases. Tracing the disk activity of ordered mode reveals that part of the data is issued to disk in sorted order based on walking the dirty page tree. The remainder, however, is issued unsorted by the commit phase as it attempts to complete all data writes for the transaction. Adding sorting to the commit phase of ordered mode solves this problem, as evidenced by the performance plotted in the graph. The rest of our performance evaluations are based on this modified version of ext3 ordered mode with sorted writing during commit.

Finally, the bottom graph in Figure 4.4 shows the slowdown of declared mode relative to ordered mode (with sorting). Overall, the performance of the two modes is extremely close, differing by no more than 3.2%.

Our next experiment tests sequential write performance to an existing 100 MB file. Figure 4.5 plots the performance of the three ext3 modes. Again, the amount written is increased along the x-axis, and fsync() is used to ensure that all data reaches disk. Ordered mode and declared mode greatly outperform data-journaling mode, achieving 22 to 23 MB/s compared to just 10 MB/s.

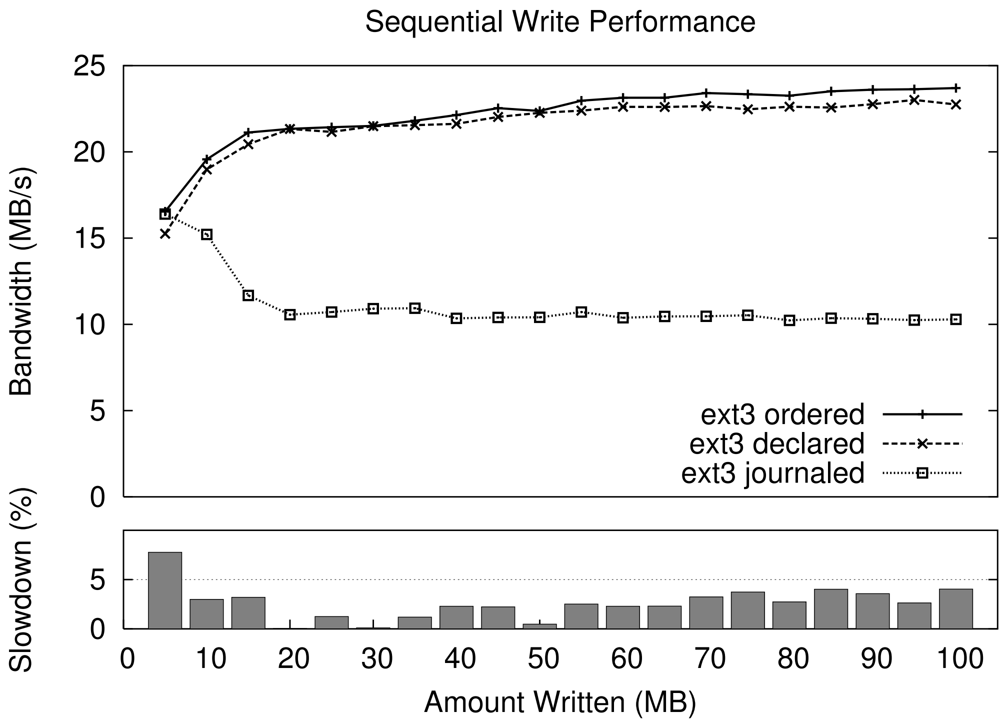


Figure 4.5 **Sequential Write Performance.** The top graph plots sequential write performance as the amount of data written is increased along the x-axis. The bottom graph shows the relative performance of declared mode as compared to ordered mode.

The bottom graph in Figure 4.5 shows the slowdown of ext3 declared mode as compared to ext3 ordered mode. Declared mode performs quite well, within 5% of ordered mode for most data points. Disk traces reveal that the performance loss is due to the fact that declared mode waits for fsync() to begin writing declare blocks and data. Because of this, ordered mode begins writing data to disk slightly earlier than declared mode. To alleviate this delay, we implement an early declare mode that begins writing declare blocks to the journal as soon as possible, that is, as soon as enough data blocks have been modified to fill a declare block. Unfortunately, this modification does not result in a performance improvement. The early writing of a few declare blocks and data blocks is offset by the seek activity between the journal and the home data locations (not shown).

Next, we examine the performance under the Sprite LFS microbenchmark [55], which creates, reads, and then unlinks a specified number of 4 KB files. Figure 4.6 plots the number of create operations completed per second as the number of files is increased along the x-axis. The bottom graph shows the slowdown of declared mode relative to ordered mode. Declared mode performs well, within 4% of ordered mode for all cases. The performance of declared mode and ordered mode are nearly identical for the other phases of the benchmark.

The ssh benchmark unpacks, configures, and builds version 2.4.0 of the ssh program from a tarred and compressed distribution file. Figure 4.7 plots the performance of each mode during the three stages of the benchmark. The execution time of each stage is normalized to that of ext3 ordered mode, and the absolute times in seconds are listed above each bar. Data-journaling mode is slighter faster than ordered mode for the configure phase, but it is 12% slower during build and

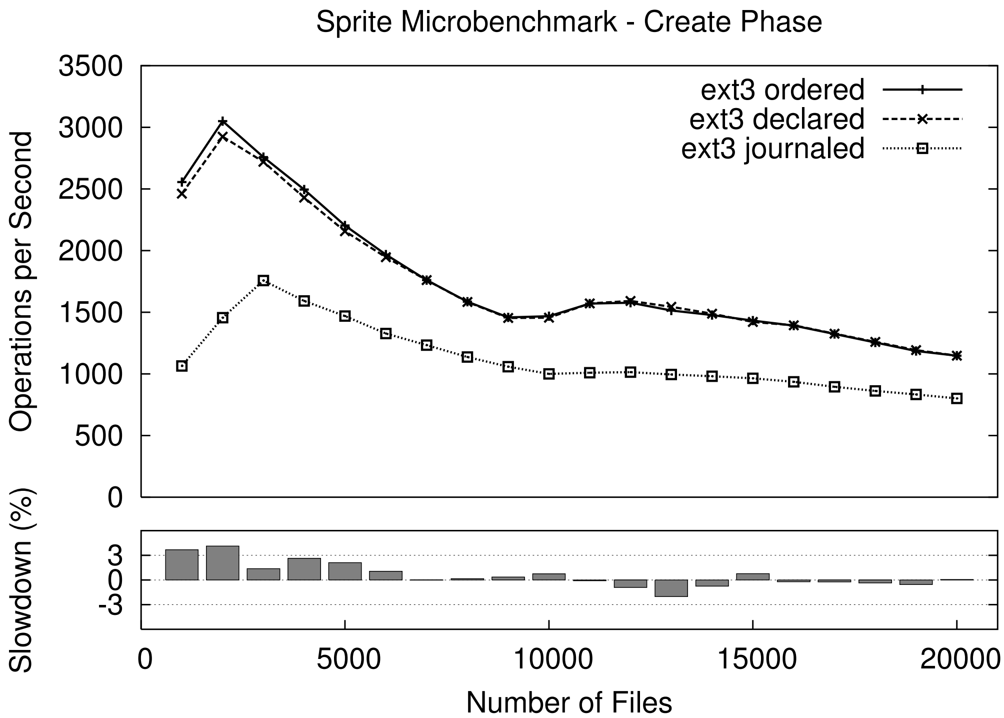


Figure 4.6 **Sprite Create Performance.** The top graph plots the performance of the create phase of the Sprite LFS microbenchmark as the number of files increases along the x-axis. The bottom graph shows the slowdown of declared mode when compared to ordered mode.

378% slower during unpack. Declared mode is quite comparable to ordered mode, running about 3% faster during unpack and configure, and 0.1% slower for the build phase.

Next, we examine ext3 performance on a modified version of the postmark benchmark that creates 5000 files across 71 directories, performs a specified number of transactions, and then deletes all files and directories. Our modification involves the addition of a call to sync() after each phase of the benchmark to ensure that data is written to disk. The unmodified version exhibits unusually high variances for all three modes of operation.

The execution time for the benchmark is shown in Figure 4.8 as the number of transactions increases along the x-axis. Data-journaling mode is extremely slow, and therefore we concentrate on the other two modes, for which we identify two interesting points. First, for large numbers of transactions, declared mode compares favorably to ordered mode, differing by approximately 5% in the worst cases. Second, with a small number of transactions, declared mode outperforms ordered mode by up to 40%. Again, disk traces help to reveal the reason. Ordered mode relies on the sorting provided by the per-file dirty page trees, and therefore its write requests are scattered across the disk. In declared mode, however, the sort performed during commit has a global view of all data being written for the transaction, thus sending the write requests to the device layer in a more efficient order.

Finally, we examine the performance of a TPC-B-like workload that performs a financial transaction across three files, adds a history record to a fourth file, and commits the changes to disk by calling sync(). The execution time of the benchmark is plotted in Figure 4.9 as the number of

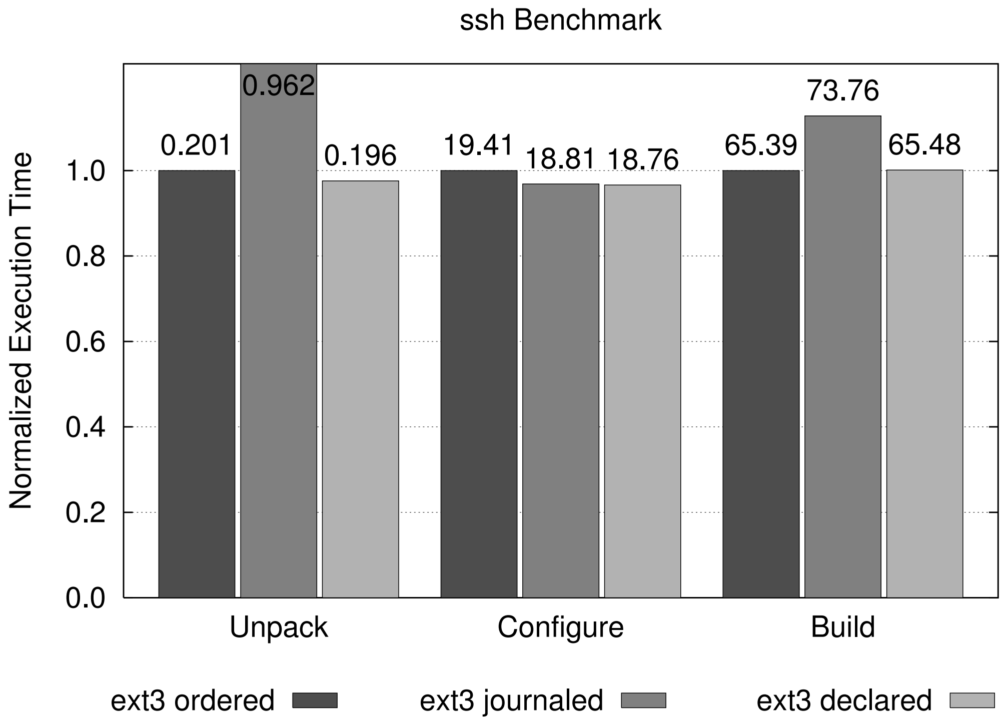


Figure 4.7 **ssh Benchmark Performance.** The graph plots the normalized execution time of the unpack, configure, and build phases of the ssh benchmark as compared to ext3 ordered mode. The absolute execution times in seconds are listed above each bar.

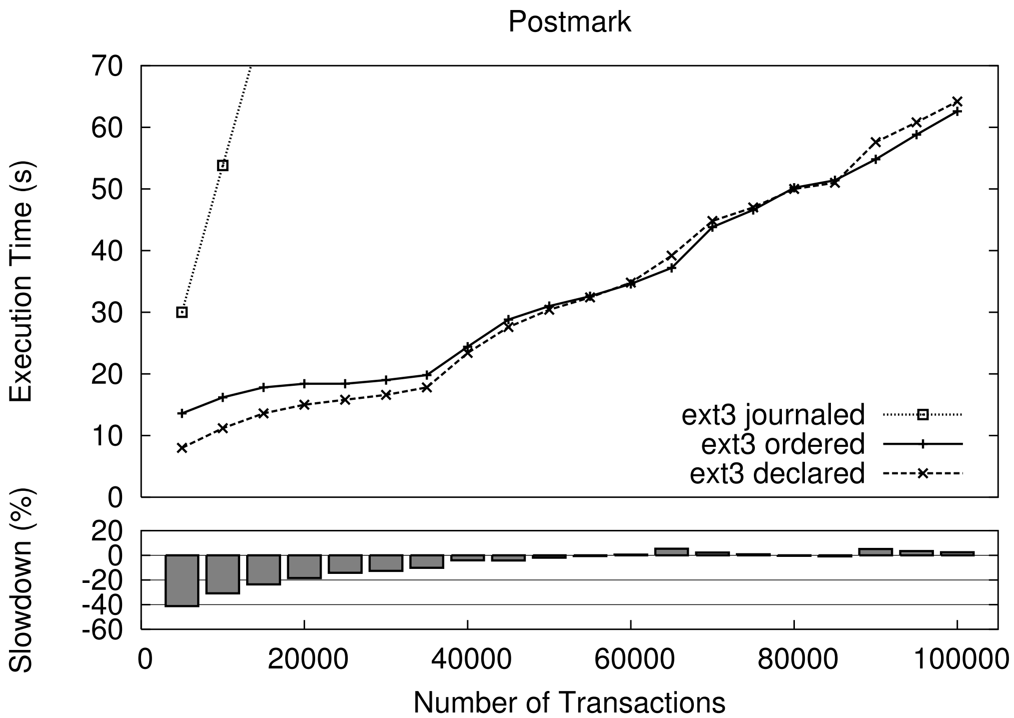


Figure 4.8 **Postmark Performance.** The top graph plots the execution time of the postmark benchmark as the number of transactions increases along the x-axis. The bottom graph shows the slowdown of declared mode when compared to ordered mode.

transactions is increased along the x-axis. In this case, declared mode consistently underperforms ext3 ordered mode by approximately 19%, and data-journaling mode performs slightly worse.

The highly synchronous nature of this benchmark presents a worst case scenario for declared mode. Each TPC-B transaction results in a very small ext3 transaction containing only four data blocks, a descriptor block, a journaled metadata block, and a commit block. The declare block at the beginning of each transaction adds 14% overhead in the number of writes performed during the benchmark. To compound this problem, the four data writes are likely serviced in parallel by the array of disks, accentuating the penalty for the declare blocks.

To examine this problem further, we test a modified version of the benchmark that forces data to disk less frequently. This has the effect of increasing the size of each application level transaction, or alternatively simulating concurrent transactions to independent data sets. Figure 4.10 shows the results of running the TPC-B benchmark with 500 transactions as the interval between calls to sync() increases along the x-axis. As the interval increases, the performance of declared mode and data-journaling mode quickly converge to that of ordered mode. Declared mode performs within 5% of ordered mode for sync() intervals of five or more transactions.

In conclusion, we find that declared mode routinely outperforms data-journaling mode. Its performance is quite close to that of ordered mode, within 5% (and sometimes better) for our random write, sequential write, and file creation microbenchmarks. It also performs within 5% of ordered mode for two macrobenchmarks, ssh and postmark. The worst performance for declared mode occurs under TPC-B with small application-level transactions, but it improves greatly as

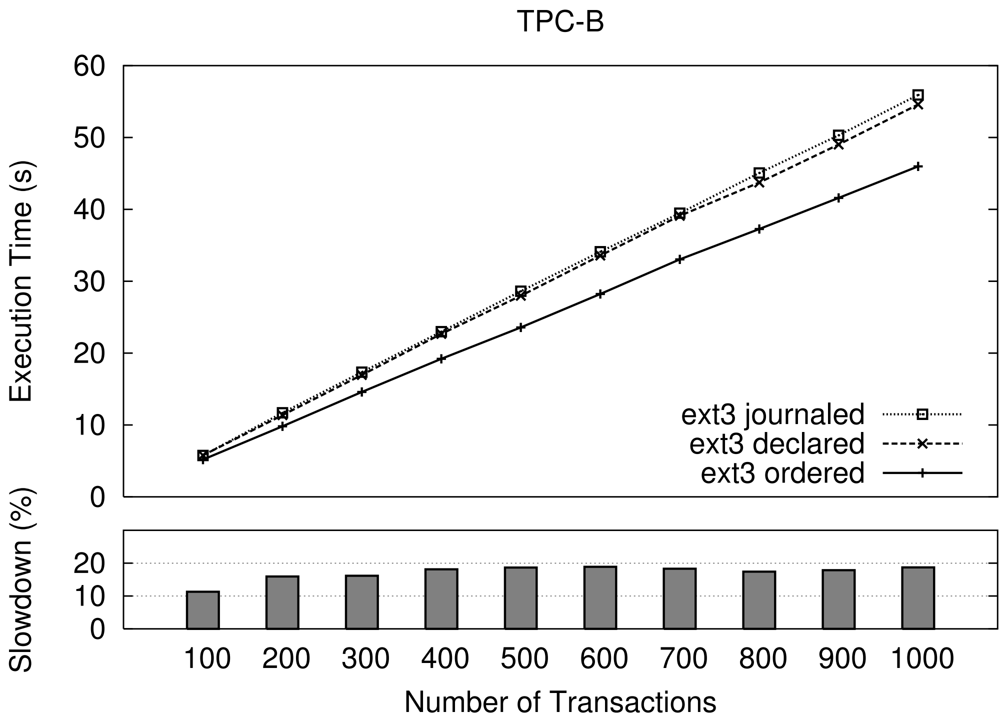


Figure 4.9 **TPC-B Performance.** The top graph plots the execution time of the TPC-B benchmark as the number of transactions increases along the x-axis. The bottom graph shows the slowdown of declared mode as compared to ordered mode.

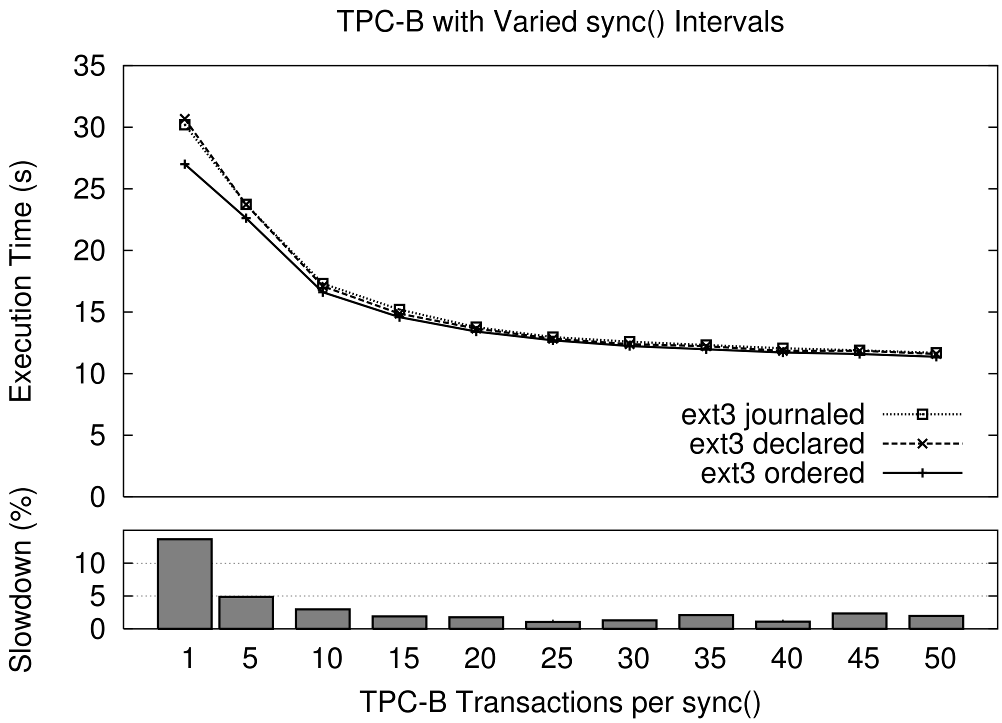


Figure 4.10 **TPC-B with Varied sync() Intervals.** The top graph plots the execution time of the TPC-B benchmark as the interval between calls to sync() increases along the x-axis. The bottom graph shows the slowdown of declared mode as compared to ordered mode.

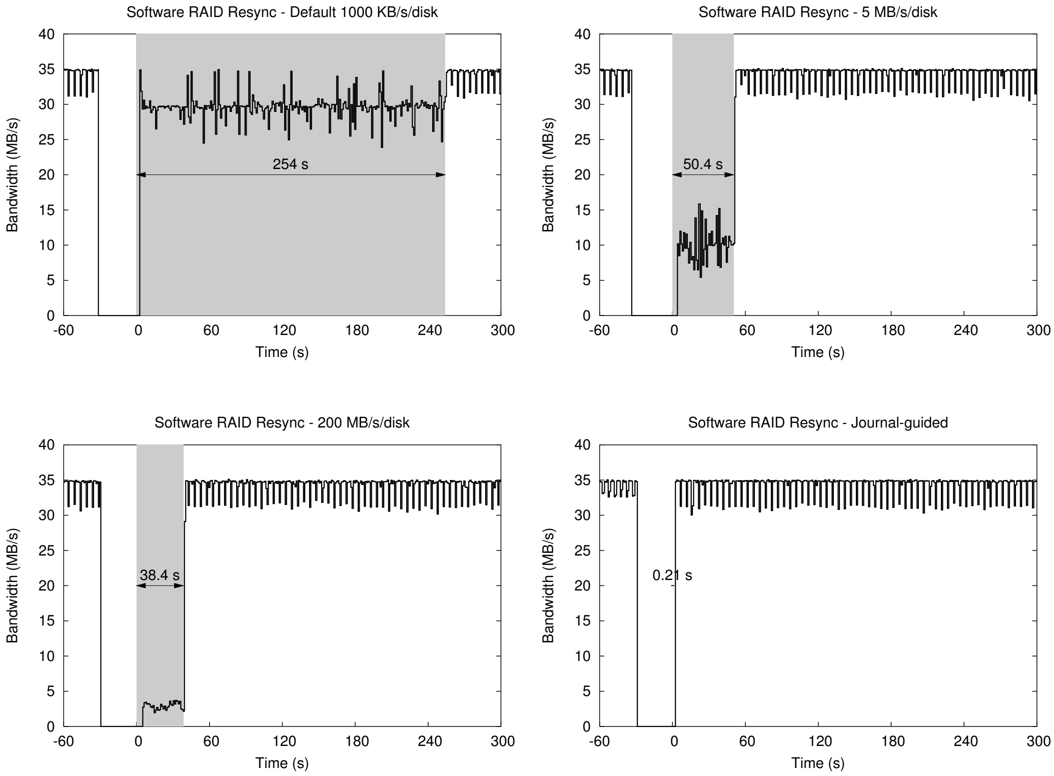
the effective transaction size increases. Overall, these results indicate that declared mode is an attractive option for enabling journal-guided resynchronization.

4.5.2 Journal-guided Resynchronization

In our final set of experiments, we examine the effect of journal-guided resynchronization. We expect a significant reduction in resync time, thus shortening the window of vulnerability and improving reliability. In addition, faster resynchronization should increase the amount of bandwidth available to foreground applications after a crash, thus improving their availability. We compare journal-guided resynchronization to the Linux software RAID resync at the default rate and at two other rates along the availability versus reliability spectrum.

The experimental workload consists of a single foreground process performing sequential reads to a set of large files. The amount of read bandwidth it achieves is measured over one second intervals. Approximately 30 seconds into the experiment, the machine is crashed and rebooted. When the machine restarts, the RAID resynchronization process begins, and the foreground process reactivates as well.

Figure 4.11 shows a series of such experiments plotting the foreground bandwidth on the y-axis as time progresses on the x-axis. Note that the origin for the x-axis coincides with the beginning of resynchronization, and the duration of the process is shaded in grey. The top left graph in the figure shows the results for the default Linux resync limit of 1000 KB/s per disk, which prefers availability over reliability. The process takes 254 seconds to scan the 1.25 GB of raw disk space in our RAID-5 array. During that time period, the foreground process bandwidth drops to 29 MB/s



Resync Type	Resync Rate Limit	Foreground Bandwidth	Vulnerability Window	Vulnerability vs. Default
Default	1000 KB/s/disk	29.58 ± 1.69 MB/s	254.00 s	100.00%
Medium	5 MB/s/disk	29.70 ± 9.48 MB/s	50.41 s	19.84%
High	200 MB/s/disk	29.87 ± 10.65 MB/s	38.44 s	15.13%
Journal-guided		34.09 ± 1.51 MB/s	0.21 s	0.08%

Figure 4.11 **Software RAID Resynchronization.** The graphs plot the bandwidth achieved by a foreground process performing sequential scans of files on a software RAID array during a system crash and the ensuing array resynchronization. The recovery period is highlighted in grey and its duration is listed. In the first three graphs, the bandwidth allocated to resynchronization is varied: the default of 1000 KB/s per disk, 5 MB/s per disk, and 200 MB/s per disk. The final graph depicts recovery using journal guidance. The table lists the availability of the foreground service and the vulnerability of the array compared to the default resynchronization period of 254 seconds following restart.

from the unimpeded rate of 34 MB/s. After resynchronization completes, the foreground process receives the full bandwidth of the array.

Linux allows the resynchronization rate to be adjusted via a sysctl variable. The top right graph in Figure 4.11 shows the effect of raising the resync limit to 5 MB/s per disk, representing a middle ground between reliability and availability. In this case, resync takes only 50.41 seconds, but the bandwidth afforded the foreground activity drops to only 9.3 MB/s. In the bottom left graph, the resync rate is set to 200 MB/s per disk, favoring reliability over availability. This has the effect of reducing the resync time to 38.44 seconds, but the foreground bandwidth drops to just 2.6 MB/s during that period.

The bottom right graph in the figure demonstrates the use of journal-guided resynchronization. Because of its knowledge of write activity before the crash, it performs much less work to correct any array inconsistencies. The process finishes in just 0.21 seconds, greatly reducing the window of vulnerability present with the previous approach. When the foreground service activates, it has immediate access to the full bandwidth of the array, increasing its availability.

The results of the experiments are summarized in the table in Figure 4.11. Each metric is calculated over the 254 second period following the restart of the machine in order to compare to the default Linux resynchronization. The 5 MB/s and 200 MB/s resync processes sacrifice availability (as seen in the foreground bandwidth variability) to improve the reliability of the array, reducing the vulnerability windows to 19.84% and 15.13% of the default, respectively. The journal-guided resync process, on the other hand, improves both the availability of the foreground process and the reliability of the array, reducing its vulnerability to just 0.08% of the default case.

Module	Orig. Lines	Mod. Lines	New Lines	Percent Change
Software RAID	3475	2	16	0.52%
ext3	8621	22	47	0.80%
Journaling	3472	43	265	8.87%
Total	15568	67	328	2.53%

Table 4.2 **Complexity of Linux Modifications.** The table lists the lines of code (counting semicolons and braces) in the original Linux 2.6.11 source and the number that were modified or added to each of the software RAID, ext3 file system, and journaling modules.

It is important to note here that the execution time of the scan-based approach scales linearly with the raw size of the array. Journal-guided resynchronization, on the other hand, is dependent only on the size of the journal, and therefore we expect it to complete in a matter of seconds even for very large arrays.

4.5.3 Complexity

Table 4.2 lists the lines of code, counted by the number of semicolons and braces, that were modified or added to the Linux software RAID, ext3 file system, and journaling modules. Very few modifications were needed to add the verify read interface to the software RAID module because the core functionality already existed and merely needed to be activated for the requested stripe. The ext3 changes involved hiding dirty buffers for declared mode and using verify reads during recovery. The majority of the changes occurred in the journaling module for writing declare blocks in the commit phase and performing careful resynchronization during recovery.

As a point of comparison, the experimental version of Linux RAID-1 bitmap logging consists of approximately 1200 lines of code, a 38% increase over RAID-1 alone. Most of our changes are to the journaling module, increasing its size by about 9%. Overall, our modifications consist of 395 lines of code, a 2.5% change across the three modules. These observations support our claim that leveraging functionality across collaborating layers can reduce the complexity of the software system.

4.6 Conclusions

We have examined the ability of a journaling file system to provide support for faster RAID resynchronization. In order to obtain a record of the outstanding writes at the time of a crash, we introduce ext3 declared mode. This new mode guarantees to declare its intentions in the journal before writing data to disk. Despite this extra write activity, declared mode performs within 5% of its predecessor.

In order to communicate this information to the RAID layer, the file system utilizes a new verify read request. This request instructs the RAID layer to read the block and repair its redundant information, if necessary. Combining these features allows us to implement fast, journal-guided resynchronization. This process improves both RAID reliability and availability by hastening the recovery process after a crash.

Our general approach advocates a system-level view for developing the storage stack as a set of collaborating layers. Using the file system journal to improve the RAID system leverages existing functionality, maintains performance, and avoids duplicating complexity in multiple components.

Chapter 5

Related Work

5.1 Gray-box Applications

Using gray-box techniques [2] to automatically uncover the behavior of underlying software and hardware layers has been explored in a number of different domains. Some of the earliest work in this area targeted the memory subsystem; for example, by measuring the time for reads of different amounts and with different strides, Saavedra and Smith reveal many interesting aspects of the memory hierarchy, including details about both caches and TLBs [58]. Similar techniques have been applied to identify aspects of a TCP protocol stack [24, 45], to determine processor cycle time [72], CPU scheduling policies [50], and buffer cache replacement policies [11].

The work most related to Shear is that which has targeted characterizing a single disk within the storage system. For example, in [90], Worthington *et al.* identify various characteristics of disks, such as the mapping of logical block numbers to physical locations, the costs of low-level operations, the size of the prefetch window, the prefetching algorithm, and the caching policy. Later, Schindler *et al.* and Talagala *et al.* build similar but more portable tools to achieve similar ends [62, 76]. We have shown how Shear can be used in conjunction with such low-level tools to discover properties of single disks inside arrays.

Semantically-Smart Disk Systems [70] take the opposite viewpoint of Shear, looking up at the interface from a storage system and inferring information about the file system. This gives the storage system an understanding of how its blocks relate to file system structures like files and directories, as well as semantic understanding of the operations occurring in the file system. This technique has been used to improve reliability [69], array caching [7], and security [68].

5.2 Storage Performance

There have been many studies of file system workloads and performance [20, 44, 53], focusing on metrics such as file access patterns, lifetimes, and caching effects. There have also been several studies of RAID performance, such as building an analytic model of a RAID [35] and determining the best stripe size for RAID-0 [14] and RAID-5 [12]. This research has largely been done in isolation, however, studying file systems on single disks or RAID systems under parameterized workloads.

Benchmarks of storage systems have usually focused on measuring performance for a given workload [9, 31, 43] rather than uncovering underlying properties, as is done by Shear. One interesting synthetic benchmark adapts its behavior to the underlying storage system [15]; this benchmark examines sensitivity to parameters such as the size of requests, the read to write ratio, and the amount of concurrency.

Livny *et al.* [37] studied the choice of clustered versus declustered file layout for synthetic database workloads on multiple disk storage systems. They found that declustered layout (striping files across disks) was preferable in most situations due to parallelism, but clustered layout (allocating a file to a single disk) was preferable under uniform access patterns and high utilization. We hope our work on Shear will promote research efforts to reevaluate these multiple disk management decisions in the context of modern file systems, disk drives, and workloads.

The HP AutoRAID [88] storage system uses a hierarchy of RAID levels beneath a logical block interface. Frequently written data blocks are stored in a mirrored region to improve performance, while infrequently written blocks are stored in RAID-5 to increase capacity. Data blocks are also migrated automatically between levels based on changes in access pattern. Like I·LFS, AutoRAID supports the addition of disk drives, and it uses log-structured writes to avoid the RAID-5 small write problem. Similar migration techniques could be supported in the I·LFS environment (akin to flexible redundancy), with the addition of file-level semantic knowledge for tracking access patterns, and greater freedom of block migration among disks.

Stodolsky *et al.* [73] examine parity logging in the RAID layer to improve the performance of small writes. Instead of writing new parity blocks directly to disk, they store a log of parity update images which are batched and written to disk in one large sequential access. Similar to NVRAM logging for resynchronization, the authors require the use of a fault tolerant buffer to store their parity update log, both for reliability and performance. These efforts to avoid small random writes support our argument that maintaining performance with RAID level logging is a complex undertaking.

The AFRAID [61] storage system overcomes the RAID-5 small write penalty be delaying the update of parity information (similar to the delayed mirroring implemented in I·LFS). This has the effect of increasing the window of vulnerability and trading reliability for improvements in performance. By adjusting the parity update policy, the system can offer a smooth transition between these qualities, and the authors find that a large performance improvement can be gained for a small reduction in reliability. Again, this system makes use of an NVRAM bitmap to record the location of stripes whose parities must be updated.

5.3 Volume Managers and Software RAID

Volume managers have long been used to ease the management of storage across multiple devices [78]. The E×RAID layer is a new type of volume manager that exposes more information to file systems (specifically, on-line performance and failure information); further, E×RAID is built with the presupposition that a single mounted file system will utilize multiple volumes for its data, whereas most volume managers assume that there is a one-to-one mapping between each mounted file system and a volume. One volume manager that is similar to E×RAID is the Pool Driver, a volume manager for SANs that has a "sub-pool" concept which may be used by a file system to group related data [77]. In that work, the GFS file system uses sub-pools to separate journaled meta-data and normal user data.

Brown and Patterson [10] examine three different software RAID systems in their work on availability benchmarks. They find that the Linux, Solaris, and Windows implementations offer differing policies during reconstruction, the process of regenerating data and parity after a disk failure. Solaris and Windows both favor reliability, while the Linux policy favors availability. Unlike our work on journal-guided resynchronization, the authors do not focus on improving the reconstruction processes, but instead on identifying the software RAID characteristics via a general benchmarking framework.

The Veritas Volume Manager [86] provides two facilities to address faster resynchronization. A dirty region log can be used to speed RAID-1 resynchronization by examining only those regions that were active before a crash. Because the log requires extra writes, however, the author warns that coarse-grained regions may be needed to maintain acceptable write performance. The Volume Manager also supports RAID-5 logging, but non-volatile memory or a solid state disk is recommended to support the extra log writes. In contrast, our ext3 declared mode offers fine-grained journal-guided resynchronization with little performance degradation and without the need for additional hardware.

5.4 Exploiting Storage Details

Part of our motivation for informing the file system (I·LFS) of the nature of the storage system is reminiscent of work on the Berkeley Fast File System (FFS) [40]. FFS is an early demonstration of the benefits of having a low-level understanding of disk technology; by co-locating correlated inodes and data blocks, performance was improved, especially as compared to the old Unix file system. Our work has the same goal, but with multi-disk storage systems in mind; however, we believe that the file system should base its decisions upon reliably-obtained information about the characteristics of storage, instead of relying upon assumptions which may or may not hold across time (*e.g.* that seek costs dominate rotational costs).

Another example of the benefits of low-level knowledge of disk characteristics is found in Schindler *et al.*'s recent work on track-aligned extents [63]. Therein, the authors explore the range of performance improvements possible when allocating and accessing data on disk-track boundaries, thereby avoiding rotational latency and track-crossing overheads in a single-disk setting. In contrast, E×RAID exposes disk boundaries of a RAID to file systems above, and not such detailed lower-level information; in the future, it would be interesting to investigate the benefits of having lower-level knowledge of the specifics of a RAID-based storage system.

Network Appliance pioneered some of the ideas we discuss here in their work on file server appliances [27]. In the development of WAFL, a write-anywhere file layout technique, Hitz *et al.* hint at how some information normally hidden inside of the RAID layer can be taken advantage of by a file system. For example, they ensure that writes to the RAID-4 layer occur in full-stripe-sized units, and thus avoid the small-write penalty that normally manifests itself on RAID-4 and RAID-5 systems. We take this a step further by formalizing the E×RAID layer, showing that a traditional file system can easily be modified to take advantage of the information provided by E×RAID, and demonstrating that a broader range of optimizations are attainable within such a framework.

5.5 Expanding Storage Interfaces

Roselli *et al.* discuss the file system/storage system gap in their talk on file system fingerprinting [54]. Their solution is to enrich the interface between file systems and storage systems, by giving the storage system more information about which blocks are related, and which blocks are likely to be accessed again in the near future. Thus, their approach gives the storage system some of the information that the file system might have collected, and presumes that the storage layer can make good use of such information. One potential problem with such an approach is that it may require agreement on a particular set of interfaces among cooperating storage vendors and file-system implementors.

Schindler *et al.* [64] augment the RAID interface to provide information about individual disks. Their Atropos volume manager exposes disk boundary and track information to provide efficient semi-sequential access to two-dimensional data structures such as database tables. The authors have since extended this work to higher dimensions [46] based on the observation that times for short seeks (tens of tracks) are roughly equivalent to the delay of seeking a single track during sequential access. Shear enables the use of such low-level information in multiple disk systems without the need for an enhanced interface.

Exposing each disk of a storage system to the file system is an extension of the arguments made by Engler and Kaashoek [22]. Therein, the authors argue that software abstractions made by operating systems are fundamentally problematic, as they are often too high-level and thus may limit power and functionality. The authors advocate a solution of exposing all hardware features to the user. Missing from this argument for minimalism is the observation that hardware itself often provides abstractions that users (and operating systems) cannot change. Apropos to data storage, the abstraction put forth by RAID systems is a particularly high-level one, which our informing interfaces break by revealing details that are often hidden from the file system.

Some distributed file systems such as Zebra [26] and xFS [1] manage each disk of the system individually, in a manner similar to I·LFS. However, both of these systems use traditional storage management techniques (such as RAID-5 striping) and do not take advantage of the many potential possibilities that the E×RAID layer makes available. In the future, we hope to extend some of our ideas into the distributed arena, and thus allow for a more direct comparison.

More recently, the NASD object interface has been introduced as a higher-level data repository for SAN-based distributed file systems [23]. This interface allows more advanced functionality to be placed into the storage layer, whereas our informing interfaces are designed to allow more functionality to be placed within the file system. Earlier work at HP on DataMesh also proposes more sophisticated interfaces for network-attached storage [87].

Our informed approach is also similar to a large body of work in parallel file systems [29, 42]. Most parallel file systems expose disk parallelism, but they allow the application itself, and not the file system, to manage it. Better control over redundancy in a parallel file system has also been proposed [18]. In that work, the computation of parity is put under user control, and in doing so, allows the user to avoid the well-known performance penalty of RAID-4 and RAID-5 under small writes.

Chapter 6

Conclusions

In this dissertation, we examined the storage stack with a goal of overcoming the information gap between file systems and storage systems stemming from the obscuring interface they share. We believe the key to overcoming this obstacle lies in information, and hence relies on the development of informing interfaces that enable vertical coordination and collaboration between layers.

6.1 Summary and Observations

In Chapter 2, we presented Shear, a system that automatically detects the important parameters of a RAID, thus transforming the obscuring logical block interface into our basic informing interface that reveals the internal structure of the array. The keys to Shear are its use of randomness to extract steady-state performance and its use of statistical techniques to deliver automated detection. We verified that Shear works as desired through a series of simulations over a variety of layout and redundancy schemes. We also showed how Shear could be used to improve the management and performance of storage arrays through its acquired information.

Overall, we found that Shear is an accurate and reliable system for uncovering RAID properties. The technique of extracting the lowest performance from the system was invaluable, though doing so meant carefully avoiding the features designed to increase performance. We imagine this technique would prove useful in characterizing other systems, as well. In general, we believe the gray-box approach is a realistic method for overcoming the limitations in current systems and demonstrating the merits of proposals for future systems.

In Chapter 3, we introduced our second informing interface, E×RAID, which built upon our basic interface to provide file system appropriate information about an array. We showed how I·LFS uses the information provided by E×RAID to bridge the gap between file systems and storage systems. We explored the implementation of on-line expansion, dynamic parallelism, flexible redundancy, and lazy mirroring. All were implemented in a relatively straight-forward manner within the file system, increasing system manageability, performance, and functionality, while maintaining a reasonable level of overall system complexity.

Some of these aspects of I·LFS would be difficult to build in the traditional storage stack. We believe this highlights our argument for vertical design and the importance of informing interfaces that allow functionality to be placed in the correct layer of the system. However, determining the proper division of labor across these layers may depend upon the metrics of importance, the properties of individual components, and the system environment. Defining interfaces that generalize to such diverse requirements will be a challenge for storage research in the future.

In Chapter 4, we took a collaborative approach in examining the ability of a journaling file system to provide support for faster software RAID resynchronization. We introduced both ext3 declared mode and the software RAID verify read interface, the combination of which allows us to implement fast, journal-guided resynchronization. This process improves both software RAID reliability and availability by hastening the recovery process after a crash, all while maintaining good performance in the common case.

Moving forward, we believe such collaborative designs will prove to be the most powerful for envisioning the storage stack of the future. Such designs could facilitate a system in which layers negotiate to define their responsibilities and actively coordinate their operations to achieve the overall goals of the system.

6.2 Future Work

6.2.1 Shear

The Shear detection process may take a long time depending on the size and particular layout of the array. To improve the runtime, it may be possible to use fewer requests during the individual microbenchmarks. The current algorithms also take a somewhat naive approach in exhaustively searching the parameter space. More refined algorithms might be able to reduce the search space based on initial findings. For example, after determining a candidate chunk size for the first disk, perhaps only the points corresponding to that estimate could be checked for the remaining disks.

The requirement of homogeneous disks limits the scope of systems that Shear can successfully examine. The key to overcoming this limitation lies in determining the pattern size over a set of heterogeneous disks. We believe the same algorithmic approach can be utilized, but the trials conducted may need to be deterministic, and the analysis phase may require modifications to establish the performance differences of the array components.

Shear also requires that it is the only process accessing the array, and this prohibits the testing of storage systems that cannot be taken offline. In the future, it may be possible to position Shear to augment an existing workload to induce the desired microbenchmarks in an online system, though doing so without severe detriment to foreground performance will be challenging.

6.2.2 Informed LFS

The current implementation of I·LFS flexible redundancy supports only striped and mirrored layouts. It would be interesting to add parity-based redundancy (similar to RAID-5) to allow for another choice in capacity and performance, and to examine the interplay of various schemes in the same file system. Prabhakaran *et al.* proposed similar parity based redundancy schemes for single disks in their work on IRON file systems [49]. Similar to the AFRAID storage system [61], I·LFS could also migrate blocks between different redundancy schemes based on file access pattern.

We also imagine that many optimizations are possible with the LFS cleaner. For example, as data is laid out on disk according to current performance characteristics and access patterns, it may not meet the needs of subsequent potentially non-sequential reads from other applications. Similarly, as new disks are added, the cleaner may want to run in order to lay out older data across the new disks. Thus, the cleaner could be used to re-organize data across drives for better read performance in the presence of heterogeneity and new drives, similar to the work of Neefe *et al.*, but generalized to operate in a heterogeneous multiple disk setting [39].

6.2.3 Journal-guided Resynchronization

Our current analysis for journal-guided resynchronization is limited to the ext3 file system for Linux. We expect the same design is applicable to other journaling file systems (such as ReiserFS [51], JFS [8], and NTFS [71]), but it would be beneficial to evaluate its complexity and performance impact in these environments. It would also be interesting to extend our system-wide view to consider opportunities for collaboration with the application layer. For example, database systems that manage their own storage typically use a form of logging to recover from failure, and therefore they could replace the file system in the role of RAID adviser.

A collaborative approach between the file system and RAID could also be used to implement *intelligent reconstruction*. When a disk fails in an array, the failed disk blocks are typically reconstructed onto a spare disk using the redundant information stored on its peers. With direction from the file system, however, the RAID layer could reconstruct only live data from the failed disk rather than blindly regenerating the entire disk, substantially lowering the time to perform the operation. Another benefit of this arrangement is that the file system could give preference to certain files over others, reconstructing higher-priority files first and thus increasing the availability of those files under failure.

6.2.4 RAID-aware File Systems

A natural extension of this dissertation research is an exploration of designing RAID-aware file systems based on the information obtained by Shear. Although I·LFS explored new file system functionality, there remain many design questions for basic file system responsibilities in the presence of multiple disks and different RAID levels.

For instance, existing operating system disk schedulers treat storage arrays as a single large disk, choosing requests based on logical block distance. In reality, those logical blocks are distributed across independent disks, a fact which could be exploited by an aware scheduler to produce a more efficient ordering.

In the realm of file layout, existing file systems still place data based on the single disk cylinder group model used by FFS [40]. A RAID-aware file system, however, could make a variety of layout decisions based on the metric of importance. If performance is paramount, the file system could explicitly distribute a file across disks to guarantee efficient use of the parallelism in the array. If the goal is reduced power consumption, the file system could place a file within an individual disk to reduce the number of active spindles required to access its content.

6.3 The End

Storage systems, and computer systems in general, are becoming more complex, yet their layers of interacting components remain concealed by abstract simplicity. Our general approach advocates a system-wide view for developing the storage stack based on greater information, enabling vertical integration across a set of collaborating layers. Though we have examined a few points in the design space of storage stacks, other arrangements are possible and perhaps preferable; we hope that they will be explored. Whatever the conclusion of research on the division of labor between file systems and storage systems, we believe the proper alignment should be arrived upon through design, implementation, and experimentation, rather than via historical artifact. Each of these layers may implement its own abstractions, protocols, mechanisms, and policies, but it is their interaction that will define the properties of the system.

LIST OF REFERENCES

[1] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, and R.Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain Resort, Colorado, December 1995.

[2] Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001.

[3] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing (Lake George), New York, October 2003.

[4] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and Dave Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data (SIGMOD '97)*, Tucson, Arizona, May 1997.

[5] Remzi H. Arpaci-Dusseau, Eric Anderson, Noah Treuhaft, David E. Culler, Joseph M. Hellerstein, Dave Patterson, and Kathy Yelick. Cluster I/O with River: Making the Fast Case Common. In *The 1999 Workshop on Input/Output in Parallel and Distributed Systems (IOPADS '99)*, Atlanta, Georgia, May 1999.

[6] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–38, Schloss Elmau, Germany, May 2001.

[7] Lakshmi N. Bairavasundaram, Muthian Sivathanu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pages 176–187, Munich, Germany, June 2004.

[8] Steve Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2000.

[9] Tim Bray. The Bonnie File System Benchmark. http://www.textuality.com/bonnie/.

[10] Aaron Brown and David A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.

[11] Nathan C. Burnett, John Bent, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, California, June 2002.

[12] Peter Chen and Edward K. Lee. Striping in a RAID Level 5 Disk Array. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 136–145, Ottawa, Canada, May 1995.

[13] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. RAID: High-performance, Reliable Secondary Storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[14] Peter M. Chen and David A. Patterson. Maximizing Performance in a Striped Disk Array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA '90)*, pages 322–331, Seattle, Washington, May 1992.

[15] Peter M. Chen and David A. Patterson. A New Approach to I/O Performance Evaluation–Self-Scaling I/O Benchmarks, Predicted I/O Performance. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 1–12, Santa Clara, California, May 1993.

[16] Paul Clements and James Bottomley. High Availability Data Replication. In *Proceedings of the 2003 Linux Symposium*, Ottawa, ON, Canada, June 2003.

[17] Douglas Comer. *Internetworking with TCP/IP Vol. 1: Principles, Protocols and Architecture*. Prentice Hall, London, 2 edition, 1991.

[18] Thomas H. Cormen and David Kotz. Integrating Theory And Practice In Parallel File Systems. In *Proceedings of the 1993 DAGS/PC Symposium (The Dartmouth Institute for Advanced Graduate Studies)*, pages 64–74, Hanover, NH, June 1993.

[19] Toni Cortes and Jesus Labarta. Extending Heterogeneity to RAID level 5. In *Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

[20] Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo I. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 203–216, San Francisco, California, April 2003.

[21] EMC Corporation. Symmetrix Enterprise Information Storage Systems. http://www.emc.com, 2002.

[22] Dawson R. Engler and M. Frans Kaashoek. Exterminate All Operating System Abstractions. In *The Fifth Workshop on Hot Topics in Operating Systems (HotOS V)*, Orcas Island, Washington, May 1995.

[23] Garth A. Gibson, David Rochberg, Jim Zelenka, David F. Nagle, Khalil Amiri, Fay W. Chang, Eugene M. Feinberg, Howard Gobioff, Chen Lee, Berend Ozceri, and Erik Riedel. File server scaling with network-attached secure disks. In *Proceedings of the 1997 Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS/PERFORMANCE '97)*, pages 272–284, Seattle, Washington, June 1997.

[24] Thomas Glaser. TCP/IP Stack Fingerprinting Principles. http://www.sans.org/newlook/resources/IDFAQ/ TCP_fingerprinting.htm, October 2000.

[25] Edward Grochowski. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. *Datatech*, September 1999.

[26] J.H. Hartman and J.K. Ousterhout. The Zebra Striped Network File System. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 29–43, Asheville, North Carolina, December 1993.

[27] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, Berkeley, CA, January 1994.

[28] Hui-I Hsiao and David DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *Proceedings of 6th International Conference on Data Engineering (ICDE '90)*, pages 456–465, Los Angeles, California, February 1990.

[29] Jay Huber, Christopher L. Elford, Daniel A. Reed, Andrew A. Chien, and David S. Blumenthal. PPFS: A High Performance Portable Parallel File System. In *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 385–394, Barcelona, Spain, July 1995.

[30] Van Jacobson. How to Kill the Internet. ftp://ftp.ee.lbl.gov/talks/vj-webflame.ps.Z, 1995.

[31] Jeffrey Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., October 1997.

[32] T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Summer. One-level Storage System. *IRE Transactions on Electronic Computers*, EC-11:223–235, April 1962.

[33] Butler W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[34] Edward K. Lee and Randy H. Katz. Performance Consequences of Parity Placement in Disk Arrays. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 190–199, Santa Clara, California, April 1991.

[35] Edward K. Lee and Randy H. Katz. An analytic performance model of disk arrays. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '93)*, pages 98–109, Santa Clara, California, May 1993.

[36] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srivivasan, and Yuanyuan Zhou. C-miner: Mining block correlations in storage systems. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 173–186, San Francisco, California, April 2004.

[37] Miron Livny, Setrag Khoshafian, and Haran Boral. Multi-disk management algorithms. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '87)*, pages 69–77, Banff, Alberta, Canada, May 1987.

[38] C. Lumb, J. Schindler, G.R. Ganger, D.F. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 87–102, San Diego, California, October 2000.

[39] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 238–251, Saint-Malo, France, October 1997.

[40] Marshall K. McKusick, William N. Joy, Sam J. Leffler, and Robert S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[41] Marshall Kirk McKusick, Willian N. Joy, Samuel J. Leffler, and Robert S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[42] Nils Nieuwejaar and David Kotz. The Galley Parallel File System. In *Proceedings of the 10th ACM International Conference on Supercomputing*, pages 374–381, Philadelphia, PA, May 1996. ACM Press.

[43] William Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[44] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[45] Jitendra Padhye and Sally Floyd. Identifying the TCP Behavior of Web Servers. In *Proceedings of SIGCOMM '01*, San Diego, California, August 2001.

[46] Steven W. Schlosserand Jiri Schindlerand Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory R. Ganger. On Multidimensional Data and Modern Disks. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 225–238, San Francisco, California, December 2005.

[47] David Patterson, Garth Gibson, and Randy Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[48] Dan Pelleg and Andrew Moore. X-means: Extending K-means with Efficient Estimation of the Number of Clusters. In *Proceedings of the 17th International Conference on Machine Learning*, June 2000.

[49] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[50] John Regehr. Inferring Scheduling Behavior with Hourglass. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[51] Hans Reiser. ReiserFS. www.namesys.com, 2004.

[52] Dennis M. Ritchie and Ken Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, July 1974.

[53] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[54] Drew Roselli, Jeanna Neefe Matthews, and Thomas E. Anderson. File System Fingerprinting. Works-In-Progress at the Third Symposium on Operating Systems Design and Implementation (OSDI '99), February 1999.

[55] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, February 1992.

[56] David S. H. Rosenthal. Evolving the Vnode Interface. In *Proceedings of the 1990 USENIX Summer Technical Conference*, pages 107–118, Anaheim, CA, 1990.

[57] Chris Ruemmler and John Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[58] Rafael H. Saavedra and Alan Jay Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

[59] Jose Renato Santos and Richard Muntz. Performance Analysis of the RIO Multimedia Storage System with Heterogeneous Disk Configurations. In *ACM Multimedia '98*, December 1998.

[60] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding When To Forget In The Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 110–123, Kiawah Island Resort, South Carolina, December 1999.

[61] Stefan Savage and John Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, pages 27–39, San Diego, California, January 1996.

[62] Jiri Schindler and Greg Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon University, November 1999.

[63] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[64] Jiri Schindler, Steven W. Schlosser, Minglong Shao, Anastassia Ailamaki, and Gregory R. Ganger. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[65] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, San Diego, California, January 1993.

[66] Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In *Proceedings of the USENIX Annual Technical Conference (USENIX '95)*, pages 249–264, New Orleans, Louisiana, January 1995.

[67] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 71–84, San Diego, California, June 2000.

[68] Muthian Sivathanu, Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[69] Muthian Sivathanu, Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, April 2004.

[70] Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, April 2003.

[71] David A. Solomon. *Inside Windows NT (Microsoft Programming Series)*. Microsoft Press, 1998.

[72] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, pages 155–166, New Orleans, Louisiana, June 1998.

[73] Daniel Stodolsky, Garth Gibson, and Mark Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, pages 64–75, San Diego, California, May 1993.

[74] Sun. Solaris Volume Manager Administration Guide. http://docs.sun.com/app/docs/doc/816-4520, July 2005.

[75] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *Proceedings of the USENIX Annual Technical Conference (USENIX '96)*, San Diego, California, January 1996.

[76] Nisha Talagala, Remzi H. Arpaci-Dusseau, and Dave Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

[77] David Teigland. The Pool Driver: A Volume Driver for SANs. Master's thesis, University of Minnesota, December 1999.

[78] David Teigland and Heinz Mauelshagen. Volume Managers in Linux. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Boston, Massachusetts, June 2001.

[79] Theodore Ts'o. http://e2fsprogs.sourceforge.net/ext2.html, June 2001.

[80] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

[81] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[82] Stephen C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[83] Robert van Renesse. Masking the Overhead of Protocol Layering. In *Proceedings of SIGCOMM '96*, pages 96–104, Stanford, California, August 1996.

[84] Elizabeth Varki, Arif Merchant, Jianzhang Xu, and Xiaozhou Qiu. Issues and challenges in the performance analysis of real disk arrays. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):559–574, June 2004.

[85] Veritas. http://www.veritas.com, June 2001.

[86] Veritas. Features of VERITAS Volume Manager for Unix and VERITAS File System. http://www.veritas.com/us/products/ volumemanager/whitepaper-02.html, July 2005.

[87] John Wilkes. DataMesh— scope and objectives: a commentary. Technical Report HP-DSD-89-44, Hewlett-Packard, July 1989.

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

[89] Theodore M. Wong and John Wilkes. My Cache or Yours? Making Storage More Exclusive. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, California, June 2002.

[90] Bruce L. Worthington, Greg R. Ganger, Yale N. Patt, and John Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *Proceedings of the 1995 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '95)*, pages 146–156, Ottawa, Canada, May 1995.

[91] Roger Zimmermann and Shahram Ghandeharizadeh. HERA: Heterogeneous Extension of RAID. Technical Report USC-CS-TR98-685, University of Southern California, 1998.