Bridging the Information Gap in Storage Protocol Stacks

Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin, Madison
{*tedenehy, dusseau, remzi*}*@cs.wisc.edu*

Abstract

The functionality and performance innovations in file systems and storage systems have proceeded largely independently from each other over the past years. The result is an information gap: *neither has information about how the other is designed or implemented, which can result in a high cost of maintenance, poor performance, duplication of features, and limitations on functionality. To bridge this gap, we introduce and evaluate a new division of labor between the storage system and the file system. We develop an enhanced storage layer known as Exposed RAID (E×RAID), which reveals information to file systems built above; specifically, E×RAID exports the parallelism and failure-isolation boundaries of the storage layer, and tracks performance and failure characteristics on a fine-grained basis. To take advantage of the information made available by E×RAID, we develop an Informed Log-Structured File System (I·LFS). I·LFS is an extension of the standard log-structured file system (LFS) that has been altered to take advantage of the performance and failure information exposed by E×RAID. 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/E×RAID 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 or impossible to implement with the traditional division of labor between file systems and storage.*

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 [27], the internals of file systems and underlying storage systems have evolved substantially, improving both performance [23] and functionality [33].

In file systems, many approaches have been developed to improve performance, including read-optimized inode and file placement [23], logging of writes [30], improved meta-data update methods [39], more scalable internal data structures [41], and off-line reorganization strategies [22]. However, almost all such techniques have been developed under the assumption that the file system will be run upon a single, traditional disk.

More recently, storage systems have also received much attention. For example, "smart" disks can improve read or write performance with block remapping techniques [11, 13, 49]. For I/O-intensive workloads, multiple-disk storage systems have been well studied in the research community [26, 51], and have achieved success in the storage industry.

These high-end storage systems provide the illusion of a single, fast disk to unsuspecting file systems above, but internally manage both parallelism and redundancy to optimize for performance, capacity, or even both [51]. Analogous to file systems, storage systems are often developed with a single (FFS-like) file system in mind.

While these changes in both file systems and parallel disk 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, each is unaware of the state the other tracks and the optimizations that the other performs.

This gap arose from a historical source: the hardware/software boundary. File systems have traditionally expected a block-based read/write interface to storage, because that interface is quite similar to what a single disk exports. With the advent of hardware-based RAID systems [26], 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.

However, the boundary between file system and storage system is changing, migrating towards a software-structuring technique rather than an interface necessitated by hardware. Software RAID drivers are available on many platforms [7], and with the advent of network-attached storage [14], client-side striping software can replace the need for hardware-based RAID systems entirely. Such software-based RAIDs are particularly attractive due to their low cost, *e.g.*, in a Linux-based system, one incurs only the cost of the machine and disks.

We term the arrangement of a file system layer on top of a software storage layer a "storage protocol stack," akin to networking protocol stacks that are prominent in communication networks [8]. There are some similarities between the two: layering is known to simplify system design, though potentially at the cost of performance [47]. 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 protocol stack, however, has not been developed in a single, coherent manner; the end result is not only poor performance but also the potential for duplication in implementation 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 [51]. Feature duplication is also a potential pitfall. For example, a log-structured file system [30] could be layered on top of a disk array that performs logging [40, 51], duplicating work and increasing system complexity unnecessarily. 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 cannot perform per-file operations, and it does not know that a block is no longer live after a file deletion, and thus cannot optimize the system in ways possible had that knowledge been available.

Thus, we believe that the time is ripe 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 protocol stack. Specifically, for each piece of storage functionality, we wish to understand where it is most easily and effectively implemented. We believe the problem is particularly germane at this time, with the move towards network-attached storage (and their proposed higher-level disk interfaces) under way [14].

In this paper, we take a first step towards our goal by exploring a single point in the spectrum of possible designs. To bridge the file system/storage system information gap, we develop and evaluate a new division of labor between the file system and storage. In this realignment, the storage layer exposes parallelism and failure isolation boundaries in part or full to file systems built above, and provides on-line performance and failure characteristics. We call this layer the *Exposed RAID* layer (E×RAID).

To take advantage of the information provided by E×RAID, we introduce an *Informed LFS* (I·LFS), an enhancement of a log-structured file system [30, 37]. By combining the performance and failure information presented by E×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×RAID is the implementation of the aforementioned benefits without a significant increase in overall complexity (and thus maintainability) of the storage protocol 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.

However, I·LFS/E×RAID is not a panacea. In particular, we find that managing redundancy within the file system can be somewhat onerous, requiring the careful placement of inodes and data blocks to ensure efficient operation under failure. Further, extending the traditional file system structure to support the enhanced functionality of I·LFS was sometimes an arduous task; perhaps a redesign of the age-old vnode layer to support informed file systems is warranted.

The rest of the paper is structured as follows. We begin with a discussion of related work in Section 2. In Section 3, we give an overview of our approach, and then we describe E×RAID and I·LFS in Sections 4 and 5, respectively. Then, in Section 6, we present an evaluation of our system. We present a discussion in Section 7, future work in Section 8, and conclude in Section 9.

2 Related Work

Part of our motivation for "informing" the file system of the nature of the storage system is reminiscent of work on the Berkeley Fast File System (FFS) [23]. 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).

Roselli *et al.* discuss the file system/storage system gap in their talk on file system fingerprinting [29]. 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.

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 [36]. 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 [16]. 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.

Volume managers have long been used to ease the management of storage across multiple devices [44]. The E×RAID layer is simply 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 [43]. In that work, the GFS file system uses sub-pools to separate journaled meta-data and normal user data.

Exposing each disk of a storage system to the file system is an extension of the arguments made by Engler and Kaashoek [12]. 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 E×RAID breaks by revealing information that is often hidden from the file system.

Some distributed file systems such as Zebra [15] 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 [14]. This interface allows more advanced functionality to be placed into the storage layer, whereas E×RAID is 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 [50].

Our informed approach is also similar to a large body of work in parallel file systems [17, 24]. 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 [9]. 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.

3 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 or impossible 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 implementing E×RAID is to facilitate the use of the information provided by E×RAID in the 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 [16, 19]. 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 [30], or subsequent research by Seltzer *et al.* [37, 38].

All of our software was developed within the context of the NetBSD 1.5 operating system. E×RAID was implemented as a set of hooks on the lower-level block-driver calls, and is described in more detail in Section 4. I·LFS was implemented by extending the NetBSD version of LFS, which is based on the original LFS for BSD Unix [37], and is described in detail in Section 5. We chose the NetBSD version of LFS as it is known to be a relatively stable and solid implementation.

4 E×RAID

We now describe the E×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.

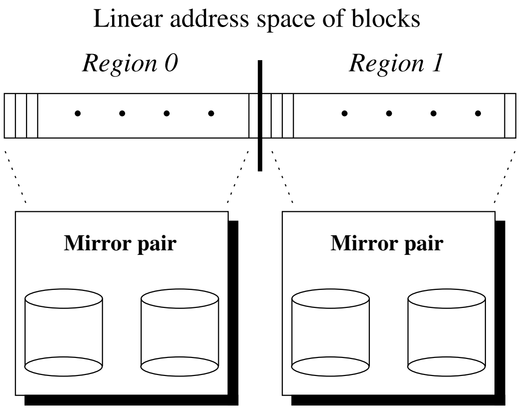

Figure 1: **An Example E×RAID Configuration.** *The diagram depicts an example E×RAID configuration in which each of two disks is combined into a mirrored pair. Two regions, each half of the size of the total address space, are presented to the client file system. Within a region, the layout performed by the mirror is hidden from the file system.*

4.1 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×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×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×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.

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 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.

4.2 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 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.

4.3 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 paper.

5 I·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.

5.1 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 [46]), 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 [48] 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 [16].

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.

5.2 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 [4, 10].

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 [3, 10, 32, 52], 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" [18].

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 [5]. 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 [4]. 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 [51]). 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.

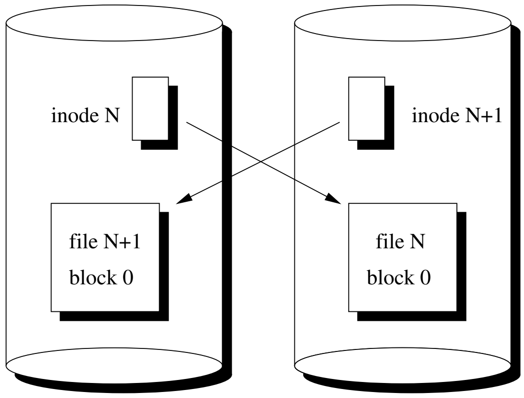

Figure 2: **The "Crossed Pointer" Problem.** *The figure illustrates the problem with using a separate file as a means for redundancy; specifically, even though each element of a file (inode, data block) has been replicated, a single lost disk could still make it difficult to find a particular data block, 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.*

5.3 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 [51], 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 [33].

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 its redundancy to one 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 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.

5.4 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 [9]. Delaying parity updates has been shown to be beneficial in RAID-5 schemes to avoid the small-write problem [34], 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 [28], 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.

6 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

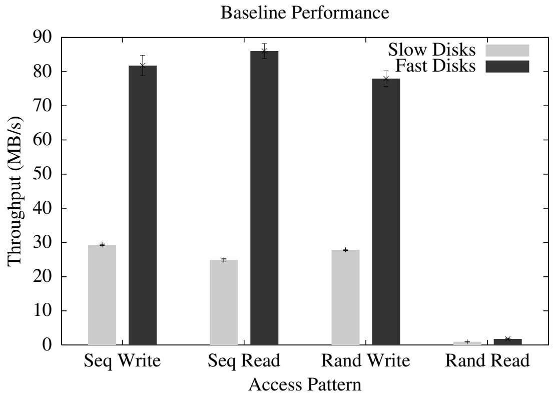

Figure 3: **Baseline Performance Comparison.** *The figure plots the performance of I·LFS/E×RAID under sequential writes, sequential reads, random writes, and random reads. The tests are run on four disks, varying whether the disks used are the four slow disks or the four fast ones. In all cases, requests generated by the tests are 8 KB in size, and the total data-set size is 200 MB.*

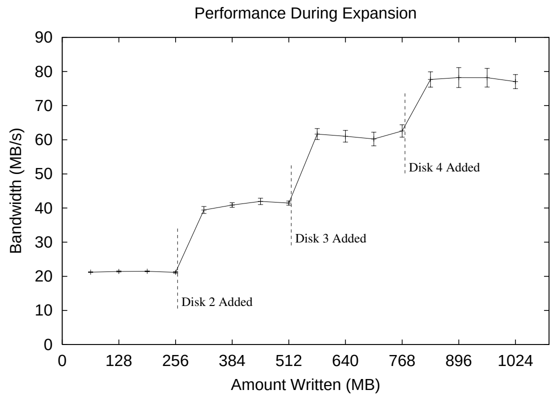

Figure 4: **Storage Expansion.** *The graph plots the performance of I·LFS during storage expansion. The experiment begins with I·LFS writing to a single disk. Each time 256 MB is written, a new disk is brought on-line, and I·LFS immediately begins writing to it for increased performance. Disk expansion is accomplished via a simple command, which adds the disk (or region) to the file system without down time.*

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.

6.1 Baseline Performance

In this first experiment, we demonstrate the baseline performance of I·LFS/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 [6] and IOzone [25] 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.

As we can see in Figure 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.

6.2 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 4 plots the performance of sequential writes over time as disks are added to the system.[1] 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.

6.3 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 [5], and therefore can also lead to performance

[1] Random writes perform similarly, due to the nature of LFS.

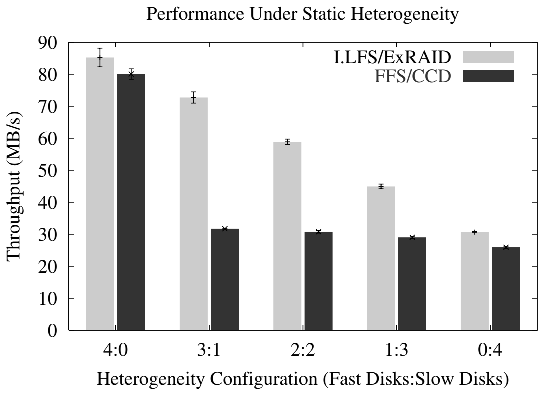

Figure 5: **Static Storage Heterogeneity.** *The figure plots the performance of I·LFS versus FFS/CCD with standard RAID-0 striping, both under a series of disk configurations. Along the x-axis, the number of fast and slow disks are varied (f:s implies f fast disks and s slow ones). By adjusting where 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.*

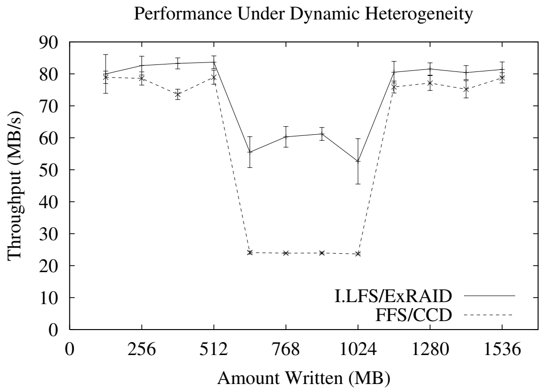

Figure 6: **Dynamic Storage Heterogeneity.** *The figure plots the performance of I·LFS/E×RAID and FFS/CCD under a dynamic performance variation. During the experiment, the performance of a single disk is temporarily degraded; the faulty disk delays requests for a fixed time, reducing throughput of the 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.*

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 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 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 [7]). 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 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.

6.4 Flexible Redundancy

In our first redundancy experiment, we verify the operation of our system in the face of failure. Figure 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

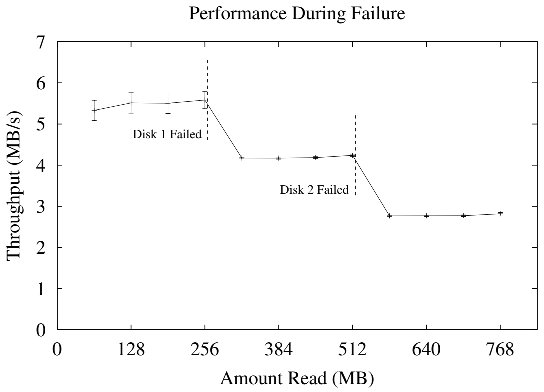

Figure 7: **Storage Failure.** *The figure plots the random read performance to a set of mirrored files across four disks on I·LFS. At the labeled points in the graph, a disk is taken off-line, and performance decreases because I·LFS can no longer balance the read load between the replicas. Note that 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.*

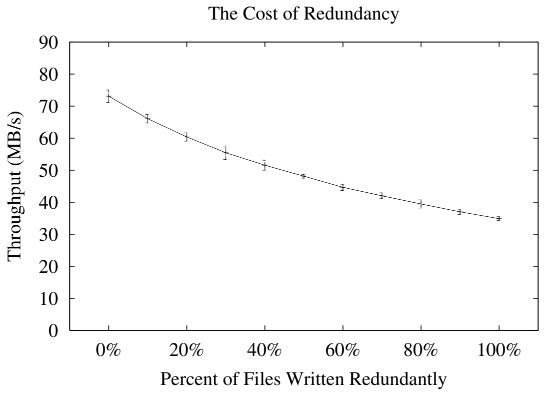

Figure 8: **Per-file Redundancy.** *The figure plots the performance of writes to 20 separate files as the percent of those files that are mirrored increases. As more files are mirrored, the net bandwidth of the system drops to roughly half of its peak rate, as expected. The peak bandwidth achieved is 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.*

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 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.

6.5 Lazy Mirroring

In our final experiment, we demonstrate some of the performance characteristics of lazy mirroring. Figure 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 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.

7 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 configu-

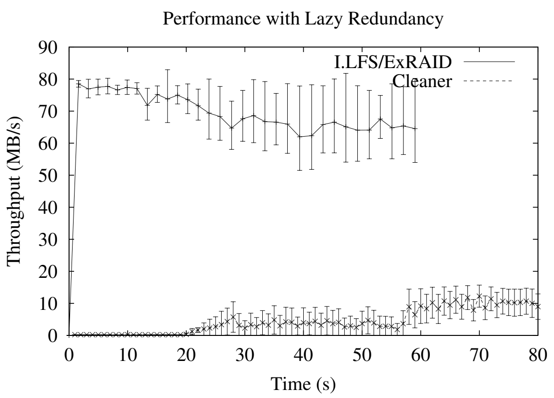

Figure 9: **Lazy Mirroring.** *The figure plots the write performance to a set of lazy redundant files on I·LFS with a replication delay of 20 seconds. Peak performance is achieved during the initial portion of the test, but performance is reduced slightly as the cleaner begins replicating data. After the write test completes, the cleaner continues to replicate data in the background.*

rations 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 [31].

8 Future Work

A number of possible avenues exist for future research. Most generally, we believe more organizations of the storage protocol stack need to be explored. Which pieces of functionality should be implemented where, and what are the trade-offs? One natural follow-on is to incorporate more lower-level information into E×RAID; the main challenge when exposing new information to the file system is to find useful pieces of information that the file system can readily exploit.

Of course, most file service today spans client and server machines. Thus, we believe it is important to consider how functionality should be split across machines. Which portion of the traditional storage protocol stack should reside on clients, and which portion should reside on the servers? Researchers in distributed file systems have taken opposing points of view on this, with systems such as Zebra [15] and xFS [1] letting clients do most of the work, whereas the Frangipani/Petal system places most functionality within the storage servers [21, 45].

We also believe cooperative approaches between the file system and storage system may be useful. For example, we found that implementing redundancy in the file system was sometimes vexing; perhaps an approach that shared the responsibility of redundancy across both file system and storage layer would be an improvement. For example, the storage layer could tell the file system which block to use as a mirror of another block, but the file system could decide when to perform the replication.

Even if we decide upon a new storage interface, it may be difficult to convince storage vendors to move away from the tried-and-true standard SCSI interface to storage. Thus, a more pragmatic approach may be to treat the RAID layer as a *gray box*, inferring its characteristics and then exploiting them in the file system, all without modification of the underlying RAID layer [2]. Tools that automatically extract low-level information from disk drives, such as DIXtrac [35] and SKIPPY [42], are first steps towards this goal, with extensions needed to understand the parallel aspects of storage systems.

Finally, we envision many more possible optimizations in our new arrangement of the storage protocol stack. For example, we are currently exploring the notion of *intelligent reconstruction*. The basic idea is simple: if a disk (or region) fails, and I·LFS has duplicated the data upon that disk, I·LFS can begin the reconstruction process itself. The key difference is that I·LFS will only reconstruct live data from that disk, and not the entire disk blindly, as a storage system would, substantially lowering the time to perform the operation. A fringe benefit of intelligent reconstruction is that I·LFS should be able to give preference to certain files over others, reconstructing higher-priority files first and thus increasing the availability of those files under failure.

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 reorganize 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 multi-disk setting [22].

9 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" [20]. In early single-disk systems, this balance was struck nearly perfectly.

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×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 paper, 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 if not impossible to build in the traditional storage protocol stack, highlighting the importance of implementing functionality in the correct layer of the system.

Though we have chosen a single point in the design space of storage protocol stacks, other arrangements are possible and perhaps even preferable; we hope that they will be explored. Whatever the conclusion of research on the division of labor between file and storage systems, we believe that the proper division should be arrived upon via design, implementation, and thorough experimentation, not via historical artifact.

10 Acknowledgements

We would like to thank our shepherd, Elizabeth Shriver, as well as John Bent, Nathan Burnett, Brian Forney, Florentina Popovici, Muthian Sivathanu, and the anonymous reviewers for their excellent feedback.

This work is sponsored by NSF CCR-0092840, CCR-0098274, NGS-0103670, CCR-0133456, ITR-0086044, and the Wisconsin Alumni Research Foundation. Timothy E. Denehy is sponsored by an NDSEG Fellowship from the Department of Defense.

References

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

[2] A. C. Arpaci-Dusseau and R. 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] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, D. E. Culler, J. M. Hellerstein, and D. Patterson. High-Performance Sorting on Networks of Workstations. In *Proceedings of the 1997 ACM SIGMOD Conference on the Management of Data (SIGMOD '97)*, pages 243–254, Tucson, AZ, May 1997.

[4] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. 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, GA, May 1999.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *The Eighth Workshop on Hot Topics in Operating Systems (HotOS VIII)*, pages 33–40, Schloss Elmau, Germany, May 2001.

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

[7] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 263–276, San Diego, CA, June 2000.

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

[9] T. H. Cormen and D. 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.

[10] T. Cortes and J. Labarta. Extending Heterogeneity to RAID level 5. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.

[11] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 15–28, Asheville, NC, December 1993.

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

[13] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 237–252, San Francisco, CA, January 1992.

[14] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka. File Server Scaling with Network-Attached Secure Disks. In *Proceedings of the 1997 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 272–284, Seattle, WA, June 1997.

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

[16] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference*, Berkeley, CA, January 1994.

[17] J. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. 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.

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

[19] 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.

[20] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 33–48, Bretton Woods, NH, December 1983. ACM.

[21] E. K. Lee and C. A. Thekkath. Petal: Distributed Virtual Disks. In *Proceedings of the Seventh Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 84–92, Cambridge, MA, October 1996.

[22] J. N. Matthews, D. Roselli, A. M. Costello, R. Y. Wang, and T. 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.

[23] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[24] N. Nieuwejaar and D. 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.

[25] W. Norcutt. The IOzone Filesystem Benchmark. http://www.iozone.org/.

[26] D. Patterson, G. Gibson, and R. 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, IL, June 1988.

[27] D. M. Ritchie and K. Thompson. The UNIX Time-Sharing System. *Comm. Assoc. Comp. Mach.*, 17(7):365–375, July 1974.

[28] D. Roselli, J. R. Lorch, and T. E. Anderson. A Comparison of File System Workloads. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 41–54, San Diego, CA, June 2000.

[29] D. Roselli, J. N. Matthews, and T. E. Anderson. File System Fingerprinting. Works-In-Progress at the Third Symposium on Operating Systems Design and Implementation (OSDI '99), February 1999.

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

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

[32] J. R. Santos and R. Muntz. Performance Analysis of the RIO Multimedia Storage System with Heterogeneous Disk Configurations. In *ACM Multimedia '98*, December 1998.

[33] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. 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, SC, December 1999.

[34] S. Savage and J. Wilkes. AFRAID — A Frequently Redundant Array of Independent Disks. In *Proceedings of the 1996 USENIX Technical Conference*, pages 27–39, San Diego, CA, January 1996.

[35] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. Technical Report CMU-CS-99-176, Carnegie Mellon, 1999.

[36] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

[37] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the 1993 USENIX Winter Technical Conference*, pages 307–326, San Diego, CA, January 1993.

[38] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging versus Clustering: A Performance Comparison. In *Proceedings of the 1995 USENIX Annual Technical Conference*, pages 249–264, New Orleans, LA, January 1995.

[39] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, CA, June 2000.

[40] D. Stodolsky, M. Holland, W. V. Courtright II, and G. A. Gibson. Parity-logging disk arrays. *ACM Transactions on Computer Systems*, 12(3):206–235, August 1994.

[41] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, CA, January 1996.

[42] N. Talagala, R. H. Arpaci-Dusseau, and D. Patterson. Microbenchmark-based Extraction of Local and Global Disk Characteristics. Technical Report CSD-99-1063, University of California, Berkeley, 1999.

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

[44] D. Teigland and H. Mauelshagen. Volume Managers in Linux. In *FREENIX Track of the USENIX 2001 Annual Technical Conference*, Boston, MA, June 2001.

[45] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 224–237, Saint-Malo, France, October 1997.

[46] T. Ts'o. http://e2fsprogs.sourceforge.net/ext2.html, June 2001.

[47] R. van Renesse. Masking the Overhead of Protocol Layering. In *Proceedings of the ACM SIGCOMM '96 Conference*, pages 96–104, Palo Alto, CA, 1996.

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

[49] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, LA, February 1999.

[50] J. Wilkes. DataMesh Research Project, Phase 1. In *Proceedings of the USENIX File Systems Workshop*, pages 63–69, May 1992.

[51] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

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