Improving Storage System Availability with D-GRAID

Muthian Sivathanu, Vijayan Prabhakaran,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison
{*muthian, vijayan, dusseau, remzi*}@*cs.wisc.edu*

Abstract

We present the design, implementation, and evaluation of D-GRAID, a gracefully-degrading and quickly-recovering RAID storage array. D-GRAID ensures that most files within the file system remain available even when an unexpectedly high number of faults occur. D-GRAID also recovers from failures quickly, restoring only live file system data to a hot spare. Both graceful degradation and live-block recovery are implemented in a prototype SCSI-based storage system underneath unmodified file systems, demonstrating that powerful "file-system like" functionality can be implemented behind a narrow block-based interface.

1 Introduction

"If a tree falls in the forest and no one hears it, does it make a sound?" *George Berkeley*

Storage systems comprised of multiple disks are the backbone of modern computing centers, and when the storage system is down, the entire center can grind to a halt. Downtime is clearly expensive; for example, in the on-line business world, millions of dollars per hour are lost when systems are not available [26, 34].

Storage system *availability* is formally defined as the mean time between failure (MTBF) divided by the sum of the MTBF and the mean time to recovery (MTTR): $\frac{MTBF}{MTBF+MTTR}$ [17]. Hence, to improve availability, one can either increase the MTBF or decrease the MTTR. Not surprisingly, researchers have studied both of these components of availability.

To increase the time between failures of a large storage array, data redundancy techniques can be applied [4, 6, 8, 18, 22, 31, 32, 33, 43, 47]. By keeping multiple copies of blocks, or through more sophisticated redundancy schemes such as parity-encoding, storage systems can tolerate a (small) fixed number of faults. To decrease the time to recovery, "hot spares" can be employed [21, 29, 32, 36]; when a failure occurs, a spare disk is activated and filled with reconstructed data, returning the system to normal operating mode relatively quickly.

However, the narrow interface between file systems and storage [13] has curtailed opportunities for improving MTBF and MTTR. In a RAID-5 storage array, if one disk too many fails before another is repaired, the entire array is corrupted. This "availability cliff" is a result of the storage system laying out blocks oblivious of their semantic importance or relationship; most files become corrupted or inaccessible after just one extra disk failure. Until a time-consuming restore from backup, the entire array remains unavailable, although most disks are still operational. Further, because the storage array has no information on which blocks are live in the file system, the recovery process must restore all blocks in the disk. This unnecessary work slows recovery and reduces availability.

An ideal storage array fails gracefully: if $\frac{1}{N}$th of the disks of the system are down, at most $\frac{1}{N}$th of the data is unavailable. An ideal array also recovers more intelligently, restoring only live data. In effect, more "important" data is less likely to disappear under failure, and such data is restored earlier during recovery. This strategy for data availability stems from Berkeley's observation about falling trees: if a file isn't available, and no process tries to access it before it is recovered, is there truly a failure?

To explore these concepts and provide a storage array with more graceful failure semantics, we present the design, implementation, and evaluation of D-GRAID, a RAID system that Degrades Gracefully (and recovers quickly). D-GRAID exploits semantic intelligence [44] within the disk array to place file system structures across the disks in a fault-contained manner, analogous to the fault containment techniques found in the Hive operating system [7] and in some distributed file systems [24, 42]. Thus, when an unexpected "double" failure occurs [17], D-GRAID continues operation, serving those files that can still be accessed. D-GRAID also utilizes semantic knowledge during recovery; specifically, only blocks that the file system considers live are restored onto a hot spare. Both aspects of D-GRAID combine to improve the effective availability of the storage array. Note that D-GRAID techniques are complementary to existing redundancy schemes; thus, if a storage administrator configures a D-GRAID array to utilize RAID Level 5, any single disk can fail without data loss, and additional failures lead to a proportional fraction of unavailable data.

In this paper, we present a prototype implementation of D-GRAID, which we refer to as *Alexander*. Alexander is an example of a semantically-smart disk system [44]. Built underneath a narrow block-based SCSI storage interface, such a disk system understands file system data structures, including the super block, allocation bitmaps, inodes, directories, and other important structures; this knowledge is central to implementing graceful degradation and quick recovery. Because of their intricate understanding of file system structures and operations, semantically-smart arrays are tailored to particular file systems; Alexander currently functions underneath unmodified Linux ext2 and VFAT file systems.

We make three important contributions to semantic disk technology. First, we deepen the understanding of how to build semantically-smart disk systems that operate correctly even with imperfect file system knowledge. Second, we demonstrate that such technology can be applied underneath widely varying file systems. Third, we demonstrate that semantic knowledge allows a RAID system to apply different redundancy techniques based on the type of data, thereby improving availability.

There are two key aspects to the Alexander implementation of graceful degradation. The first is *selective meta-data replication*, in which Alexander replicates naming and system meta-data structures of the file system to a high degree while using standard redundancy techniques for data. Thus, with a small amount of overhead, excess failures do not render the entire array unavailable. Instead, the entire directory hierarchy can still be traversed, and only some fraction of files will be missing, proportional to the number of missing disks. The second is a *fault-isolated data placement* strategy. To ensure that semantically meaningful data units are available under failure, Alexander places semantically-related blocks (*e.g.*, the blocks of a file) within the storage array's unit of fault-containment (*e.g.*, a disk). By observing the natural failure boundaries found within an array, failures make semantically-related groups of blocks unavailable, leaving the rest of the file system intact.

Unfortunately, fault-isolated data placement improves availability at a cost; related blocks are no longer striped across the drives, reducing the natural benefits of parallelism found within most RAID techniques [15]. To remedy this, Alexander also implements *access-driven diffusion* to improve throughput to frequently-accessed files, by spreading a copy of the blocks of "hot" files across the drives of the system. Alexander monitors access to data to determine which files to replicate in this fashion, and finds space for those replicas either in a pre-configured *performance reserve* or opportunistically in the unused portions of the storage system.

We evaluate the availability improvements possible with D-GRAID through trace analysis and simulation, and find that D-GRAID does an excellent job of masking an arbitrary number of failures from most processes by enabling continued access to "important" data. We then evaluate our prototype Alexander under microbenchmarks and trace-driven workloads. We find that the construction of D-GRAID is feasible; even with imperfect semantic knowledge, powerful functionality can be implemented within a block-based storage array. We also find that the run-time overheads of D-GRAID are small, but that the CPU costs as compared to a standard array are high. We show that access-driven diffusion is crucial for performance, and that live-block recovery is effective when disks are under-utilized. The combination of replication, data placement, and recovery techniques results in a storage system that improves availability while maintaining a high level of performance.

The rest of this paper is structured as follows. In Section 2, we present extended motivation, and in Section 3, we discuss the design principles of D-GRAID. In Section 4, we present trace analysis and simulations, and discuss semantic knowledge in Section 5. In Section 6, we present our prototype implementation. We evaluate our prototype in Section 7, discuss alternative methods to implementing D-GRAID and the commercial feasibility of a semantic disk based approach in Section 8. In Section 9, we present related work and conclude in Section 10.

2 Extended Motivation

The Case for Graceful Degradation: RAID redundancy techniques typically export a simple failure model. If D or fewer disks fail, the RAID continues to operate correctly, but perhaps with degraded performance. If more than D disks fail, the RAID is entirely unavailable until the problem is corrected, perhaps via a restore from tape. In most RAID schemes, D is small (often 1); thus even when most disks are working, users may observe a "failed" disk system.

With graceful degradation, a RAID system can absolutely tolerate some fixed number of faults (as before), and excess failures are not catastrophic; most of the data (an amount proportional to the number of disks still available in the system) continues to be available, thus allowing access to that data while the other "failed" data is restored. It does not matter to users or applications whether the entire contents of the volume are present; rather, what matters is whether a particular set of files are available.

One question is whether it is realistic to expect a catastrophic failure scenario within a RAID system. For example, in a RAID-5 system, given the high MTBF's reported by disk manufacturers, one might believe that a second disk failure is highly unlikely to occur before the first failed disk is repaired. However, multiple disk failures do occur, for two primary reasons. First, correlated faults are more common in systems than expected [19]. If the RAID has not been carefully designed in an orthogonal manner, a single controller fault or other component error can render a fair number of disks unavailable [8]; such redundant designs are expensive, and therefore may only be found in higher end storage arrays. Second, Gray points out that system administration is the main source of failure in systems [17]. A large percentage of human failures occur during maintenance, where "the maintenance person typed the wrong command or unplugged the wrong module, thereby introducing a double failure" (page 6) [17].

Other evidence also suggests that multiple failures can occur. For example, IBM's ServeRAID array controller product includes directions on how to attempt data recovery when multiple disk failures occur within a RAID-5 storage array [23]. Within our own organization, data is stored on file servers under RAID-5. In one of our servers, a single disk failed, but the indicator that should have informed administrators of the problem did not do so. The problem was only discovered when a second disk in the array failed; full restore from backup ran for days. In this scenario, graceful degradation would have enabled access to a large fraction of user data during the long restore.

One might think that the best approach to dealing with multiple failures would be to employ a higher level of redundancy [2, 6], thus enabling the storage array to tolerate a greater number of failures without loss of data. However, these techniques are often expensive (*e.g.*, three-way data mirroring) or bandwidth-intensive (*e.g.*, more than 6 I/Os per write in a P+Q redundant store). Graceful degradation is complementary to such techniques. Thus, storage administrators could choose the level of redundancy they believe necessary for common case faults; graceful degradation is enacted when a "worse than expected" fault occurs, mitigating its ill effect.

Need for Semantically-Smart Storage: Implementing new functionality in a semantically-smart disk system has the key benefit of enabling wide-scale deployment underneath an unmodified SCSI interface without any OS modification, thus working smoothly with existing file systems and software base. Although there is some desire to evolve the interface between file systems and storage [16], the reality is that current interfaces will likely survive much longer than anticipated. As Bill Joy once said, "systems may come and go, but protocols live forever". A new mechanism like D-GRAID is more likely to be deployed if it is non-intrusive on existing infrastructure; semantic disks ensure just that.

3 Design: D-GRAID Expectations

In this section, we discuss the design of D-GRAID. We present background information on file systems, the data layout strategy required to enable graceful degradation, the important design issues that arise due to the new layout, and the process of fast recovery.

3.1 File System Background

Semantic knowledge is system specific; therefore, we discuss D-GRAID design and implementation for two widely differing file systems: Linux ext2 [45] and Microsoft VFAT [30] file system. Inclusion of VFAT represents a significant contribution compared to previous research, which operated solely underneath UNIX file systems.

The ext2 file system is an intellectual descendant of the Berkeley Fast File System (FFS) [28]. The disk is split into a set of *block groups*, akin to cylinder groups in FFS, each of which contains bitmaps to track inode and data block allocation, inode blocks, and data blocks. Most information about a file, including size and block pointers, are found in the file's inode.

The VFAT file system descends from the world of PC operating systems. In this paper, we consider the Linux VFAT implementation of FAT-32, although our work is general and applies to other variants. VFAT operations are centered around the eponymous file allocation table, which contains an entry for each allocatable block in the file system. These entries are used to locate the blocks of a file, in a linked-list fashion, *e.g.*, if a file's first block is at address b, one can look in entry b of the FAT to find the next block of the file, and so forth. An entry can also hold an end-of-file marker or a setting that indicates the block is free. Unlike UNIX file systems, where most information about a file is found in its inode, a VFAT file system spreads this information across the FAT itself and the directory entries; the FAT is used to track which blocks belong to the file, whereas the directory entry contains information like size, permission, and type information.

3.2 Graceful Degradation

To ensure partial availability of data under multiple failures in a RAID array, D-GRAID employs two main techniques. The first is a *fault-isolated data placement* strategy, in which D-GRAID places each "semantically-related set of blocks" within a "unit of fault containment" found within the storage array. For simplicity of discussion, we assume that a file is a semantically-related set of blocks, and that a single disk is the unit of fault containment. We will generalize the former below, and the latter is easily generalized if there are other failure boundaries that should be observed (*e.g.*, SCSI chains). We refer to the physical disk to which a file belongs as the *home site* for the file. When a particular disk fails, fault-isolated data placement ensures that only files that have that disk

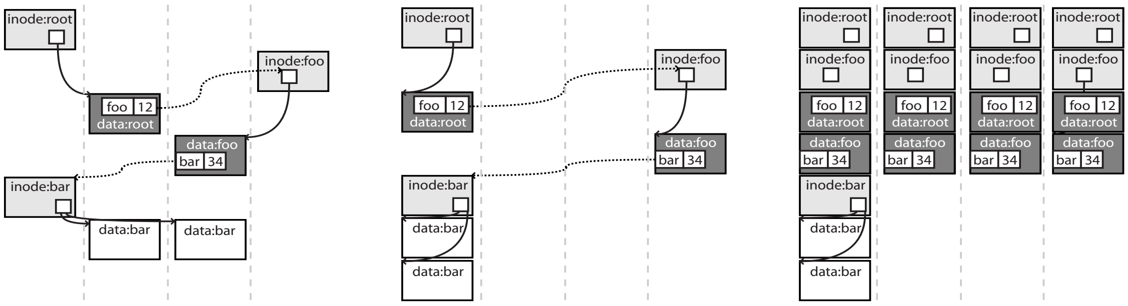

Figure 1: **A Comparison of Layout Schemes.** *These figures depict different layouts of a file "/foo/bar" in a* UNIX *file system starting at the root inode and following down the directory tree to the file data. Each vertical column represents a disk. For simplicity, the example assumes no data redundancy for user file data. On the left is a typical file system layout on a non-D-GRAID disk system; because blocks (and therefore pointers) are spread throughout the file system, any single fault will render the blocks of the file "bar" inaccessible. In the middle is a fault-isolated data placement of files and directories. In this scenario, if one can access the inode of a file, one can access its data (indirect pointer blocks would also be constrained within the same disk). Finally, on the right is an example of selective meta-data replication. By replicating directory inodes and directory blocks, D-GRAID can guarantee that users can get to all files that are available. Some of the requisite pointers have been removed from the rightmost figure for simplicity. Color codes are white for user data, light shaded for inodes, and dark shaded for directory data.*

as their home site become unavailable, while other files remain accessible as whole files.

The second technique is *selective meta-data replication*, in which D-GRAID replicates naming and system meta-data structures of the file system to a high degree, *e.g.*, directory inodes and directory data in a UNIX file system. D-GRAID thus ensures that all live data is reachable and not orphaned due to failure. The entire directory hierarchy remains traversable, and the fraction of missing user data is proportional to the number of failed disks.

Thus, D-GRAID lays out logical file system blocks in such a way that the availability of a single file depends on as few disks as possible. In a traditional RAID array, this dependence set is normally the entire set of disks in the group, thereby leading to entire file system unavailability under an unexpected failure. A UNIX-centric example of typical layout, fault-isolated data placement, and selective meta-data replication is depicted in Figure 1. Note that for the techniques in D-GRAID to work, a meaningful subset of the file system must be laid out within a single D-GRAID array. For example, if the file system is striped across multiple D-GRAID arrays, no single array will have a meaningful view of the file system. In such a scenario, D-GRAID can be run at the logical volume manager level, viewing each of the arrays as a single disk; the same techniques remain relevant.

Because D-GRAID treats each file system block type differently, the traditional RAID taxonomy is no longer adequate in describing how D-GRAID behaves. Instead, a finer-grained notion of a RAID level is required, as D-GRAID may employ different redundancy techniques for different types of data. For example, D-GRAID commonly employs n-way mirroring for naming and system meta-data, whereas it uses standard redundancy techniques, such as mirroring or parity encoding (*e.g.*, RAID-5), for user data. Note that n, a value under administrative control, determines the number of failures under which D-GRAID will degrade gracefully. In Section 4, we will explore how data availability degrades under varying levels of namespace replication.

3.3 Design Considerations

The layout and replication techniques required to enable graceful degradation introduce a host of design issues. We highlight the major challenges that arise.

Semantically-related blocks: With fault-isolated data placement, D-GRAID places a logical unit of file system data (*e.g.*, a file) within a fault-isolated container (*e.g.*, a disk). Which blocks D-GRAID considers "related" thus determines which data remains available under failure. The most basic approach is *file-based* grouping, in which a single file (including its data blocks, inode, and indirect pointers) is treated as the logical unit of data; however, with this technique a user may find that some files in a directory are unavailable while others are not, which may cause frustration and confusion. Other groupings preserve more meaningful portions of the file system volume under failure. With *directory-based* grouping, D-GRAID ensures that the files of a directory are all placed within the same unit of fault containment. Less automated options are also possible, allowing users to specify arbitrary semantic groupings which D-GRAID then treats as a unit.

Load balance: With fault-isolated placement, instead of placing blocks of a file across many disks, the blocks are isolated within a single home site. Isolated placement improves availability but introduces the problem of load balancing, which has both space and time components.

In terms of space, the total utilized space in each disk should be maintained at roughly the same level, so that when a fraction of disks fail, roughly the same fraction of data becomes unavailable. Such balancing can be addressed in the foreground (*i.e.*, when data is first allocated), the background (*i.e.*, with migration), or both. Files (or directories) larger than the amount of free space in a single disk can be handled either with a potentially expensive reorganization or by reserving large extents of free space on a subset of drives. Files that are larger than a single disk must be split across disks.

More pressing are the performance problems introduced by fault-isolated data placement. Previous work indicates that striping of data across disks is better for performance even compared to sophisticated file placement algorithms [15, 48]. Thus, D-GRAID makes additional copies of user data that are spread across the drives of the system, a process which we call *access-driven diffusion*. Whereas standard D-GRAID data placement is optimized for availability, access-driven diffusion increases performance for those files that are frequently accessed. Not surprisingly, access-driven diffusion introduces policy decisions into D-GRAID, including where to place replicas that are made for performance, which files to replicate, and when to create the replicas.

Meta-data replication level: The degree of meta-data replication within D-GRAID determines how resilient it is to excessive failures. Thus, a high degree of replication is desirable. Unfortunately, meta-data replication comes with costs, both in terms of space and time. For space overheads, the trade-offs are obvious: more replicas imply more resiliency. One difference between traditional RAID and D-GRAID is that the amount of space needed for replication of naming and system meta-data is dependent on usage, *i.e.*, a volume with more directories induces a greater amount of overhead. For time overheads, a higher degree of replication implies lowered write performance for naming and system meta-data operations. However, others have observed that there is a lack of update activity at higher levels in the directory tree [35], and lazy update propagation can be employed to reduce costs [43].

3.4 Fast Recovery

Because the main design goal of D-GRAID is to ensure higher availability, fast recovery from failure is also critical. The most straightforward optimization available with D-GRAID is to recover only "live" file system data. Assume we are restoring data from a live mirror onto a hot spare; in the straightforward approach, D-GRAID simply scans the source disk for live blocks, examining appropriate file system structures to determine which blocks to restore. This process is readily generalized to more complex redundancy encodings. D-GRAID can potentially prioritize recovery in a number of ways, *e.g.*, by restoring certain "important" files first, where importance could be domain specific (*e.g.*, files in `/etc`) or indicated by users in a manner similar to the hoarding database in Coda [27].

4 Exploring Graceful Degradation

In this section, we use simulation and trace analysis to evaluate the potential effectiveness of graceful degradation and the impact of different semantic grouping techniques. We first quantify the space overheads of D-GRAID. Then we demonstrate the ability of D-GRAID to provide continued access to a proportional fraction of meaningful data after arbitrary number of failures. More importantly, we then demonstrate how D-GRAID can hide failures from users by replicating "important" data. The simulations use file system traces collected from HP Labs [38], and cover 10 days of activity; there are 250 GB of data spread across 18 logical volumes.

	Level of Replication		
	1-way	4-way	16-way
ext2$_{1KB}$	0.15%	0.60%	2.41%
ext2$_{4KB}$	0.43%	1.71%	6.84%
VFAT$_{1KB}$	0.52%	2.07%	8.29%
VFAT$_{4KB}$	0.50%	2.01%	8.03%

Table 1: **Space Overhead of Selective Meta-data Replication.** *The table shows the space overheads of selective meta-data replication as a percentage of total user data, and as the level of naming and system meta-data replication increases. In the leftmost column, the percentage space overhead without any meta-data replication is shown. The next two columns depict the costs of modest (4-way) and paranoid (16-way) schemes. Each row shows the overhead for a particular file system, either ext2 or VFAT, with block size set to 1 KB or 4 KB.*

4.1 Space Overheads

We first examine the space overheads due to selective meta-data replication that are typical with D-GRAID-style redundancy. We calculate the cost of selective meta-data replication as a percentage overhead, measured across all volumes of the HP trace data. We calculate the highest selective meta-data replication overhead percentage possible by assuming no replication of user data; if user data is mirrored, the overheads are cut in half.

Table 1 shows that selective meta-data replication induces only a mild space overhead even under high levels of meta-data redundancy for both the Linux ext2 and VFAT file systems. Even with 16-way redundancy of meta-data, only a space overhead of 8% is incurred in the worst case (VFAT with 1 KB blocks). With increasing block size, while ext2 uses more space (due to internal fragmentation with larger directory blocks), the overheads actually decrease with VFAT. This phenomenon is due to the structure of VFAT; for a fixed-sized file system, as block size grows, the file allocation table itself shrinks, although the blocks that contain directory data grow.

4.2 Static Availability

We next examine how D-GRAID availability degrades under failure with two different semantic grouping strategies. The first strategy is file-based grouping, which keeps the information associated with a single file within a failure boundary (*i.e.*, a disk); the second is directory-based grouping, which allocates files of a directory together. For this analysis, we place the entire 250 GB of files and directories from the HP trace onto a simulated 32-disk sys-

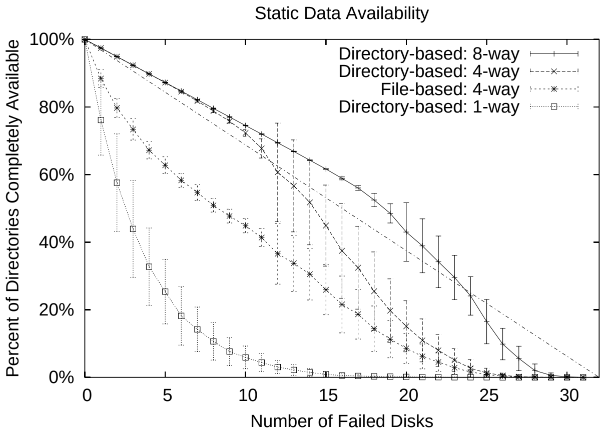

Figure 2: **Static Data Availability.** *The percent of entire directories available is shown under increasing disk failures. The simulated system consists of 32 disks, and is loaded with the 250 GB from the HP trace. Two different strategies for semantic grouping are shown: file-based and directory-based. Each line varies the level of replication of namespace meta-data. Each point shows average and deviation across 30 trials, where each trial randomly varies which disks fail.*

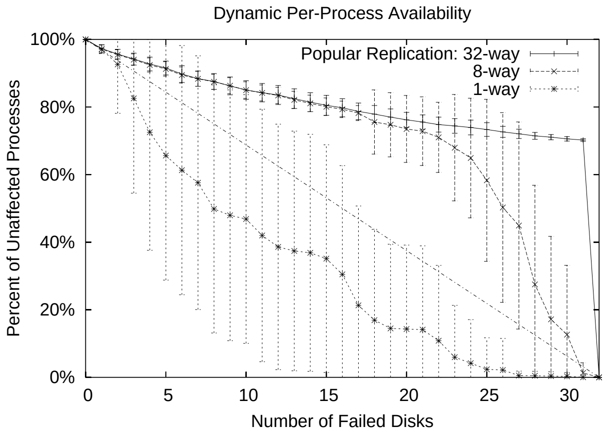

Figure 3: **Dynamic Data Availability.** *The figure plots the percent of processes that run unaffected under disk failure from one busy hour from the HP trace. The degree of namespace replication is set aggressively to 32. Each line varies the amount of replication for "popular" directories; 1-way implies that those directories are not replicated, whereas 8-way and 32-way show what happens with a modest and extreme amount of replication. Means and deviations of 30 trials are shown.*

tem, remove simulated disks, and measure the percentage of whole directories that are available. We assume no user data redundancy (*i.e.*, D-GRAID Level 0).

Figure 2 shows the percent of directories available, where a directory is available if all of its files are accessible (although subdirectories and their files may not be). From the figure, we observe that graceful degradation works quite well, with the amount of available data proportional to the number of working disks, in contrast to a traditional RAID where a few disk crashes would lead to complete data unavailability. In fact, availability sometimes degrades slightly less than expected from a strict linear fall-off; this is due to a slight imbalance in data placement across disks and within directories. Further, even a modest level of namespace replication (*e.g.*, 4-way) leads to very good data availability under failure. We also conclude that with file-based grouping, some files in a directory are likely to "disappear" under failure, leading to user dissatisfaction.

4.3 Dynamic Availability

Finally, by simulating dynamic availability, we examine how often users or applications will be oblivious that D-GRAID is operating in degraded mode. Specifically, we run a portion of the HP trace through a simulator with some number of failed disks, and record what percent of processes observed no I/O failure during the run. Through this experiment, we find that namespace replication is not enough; certain files, that are needed by most processes, must be replicated as well.

In this experiment, we set the degree of namespace replication to 32 (full replication), and vary the level of replication of the contents of popular directories, *i.e.*, `/usr/bin`, `/bin`, `/lib` and a few others. Figure 3 shows that without replicating the contents of those directories, the percent of processes that run without ill-effect is lower than expected from our results in Figure 2. However, when those few directories are replicated, the percentage of processes that run to completion under disk failure is much better than expected. The reason for this is clear: a substantial number of processes (*e.g.*, `who`, `ps`, etc.) only require that their executable and a few other libraries are available to run correctly. With popular directory replication, excellent availability under failure is possible. Fortunately, almost all of the popular files are in "read only" directories; thus, wide-scale replication will not raise write performance or consistency issues. Also, the space overhead due to popular directory replication is minimal for a reasonably sized file system; for this trace, such directories account for about 143 MB, less than 0.1% of the total file system size.

5 Semantic Knowledge

We now move towards the construction of a D-GRAID prototype underneath a block-based SCSI-like interface. The enabling technology underlying D-GRAID is semantic knowledge [44]. Understanding how the file system above utilizes the disk enables D-GRAID to implement both graceful degradation under failure and quick recovery. The exact details of acquiring semantic knowledge within a disk or RAID system have been described elsewhere [44]; here we just assume that a basic understanding of file system layout and structures is available within the storage system. Specifically, we assume that D-GRAID has static knowledge of file system layout, including which regions on disk are used for which block types and the contents of specific block types, *e.g.*, the fields of an inode.

5.1 File System Behaviors

In this paper, we extend understanding of semantically-smart disks by presenting techniques to handle more general file system behaviors. Previous work required the file system to be mounted synchronously when implementing complex functionality within the disk; we relax that requirement. We now describe our assumptions for general file system behavior; we believe that many, if not all, modern file systems adhere to these behavioral guidelines.

First, blocks in a file system can be dynamically typed, *i.e.*, the file system can locate different types of blocks at the same physical location on disk over the lifetime of the file system. For example, in a UNIX file system, a block in the data region can be a user-data block, an indirect-pointer block or a directory-data block. Second, a file system can delay updates to disk; delayed writes at the file system facilitate batching of small writes in memory and suppressing of writes to files that are subsequently deleted. Third, as a consequence of delayed writes, the order in which the file system actually writes data to disk can be arbitrary. Although certain file systems order writes carefully [14], to remain general, we do not make any such assumptions on the ordering. Note that our assumptions are made for practical reasons: the Linux ext2 file system exhibits all the aforementioned behaviors.

5.2 Accuracy of Information

Our assumptions about general file system behavior imply that the storage system cannot accurately classify the type of each block. Block classification is straightforward when the type of the block depends upon its location on disk. For example, in the Berkeley Fast File System (FFS) [28], the regions of disk that store inodes are fixed at file system creation; thus, any traffic to those regions is known to contain inodes.

However, type information is sometimes spread across multiple blocks. For example, a block filled with indirect pointers can only be identified as such by observing the corresponding inode, specifically that the inode's indirect pointer field contains the address of the given indirect block. More formally, to identify an indirect block B, the semantic disk must look for the inode that has block B in its indirect pointer field. Thus, when the relevant inode block I_B is written to disk, the disk infers that B is an indirect block, and when it later observes block B written, it uses this information to classify and treat the block as an indirect block. However, due to the delayed write and reordering behavior of the file system, it is possible that in the time between the disk writes of I_B and B, block B was freed from the original inode and was reallocated to another inode with a different type, *i.e.*, as a normal data block. The disk does not know this since the operations took place in memory and were not reflected to disk. Thus, the inference made by the semantic disk on the block type could be wrong due to the inherent staleness of the information tracked. Implementing a correct system despite potentially inaccurate inferences is one of the challenges we address in this paper.

6 Implementation: Making D-GRAID

We now discuss the prototype implementation of D-GRAID known as Alexander. Alexander uses fault-isolated data placement and selective meta-data replication to provide graceful degradation under failure, and employs access-driven diffusion to correct the performance problems introduced by availability-oriented layout. Currently, Alexander replicates namespace and system meta-data to an administrator-controlled value (*e.g.*, 4 or 8), and stores user data in either a RAID-0 or RAID-1 manner; we refer to those systems as D-GRAID Levels 0 and 1, respectively. We are currently pursuing a D-GRAID Level 5 implementation, which uses log-structuring [39] to avoid the small-write problem that is exacerbated by fault-isolated data placement.

In this section, we present the implementation of graceful degradation and live-block recovery, with most of the complexity (and hence discussion) centered around graceful degradation. For simplicity of exposition, we focus on the construction of Alexander underneath the Linux ext2 file system. At the end of the section, we discuss differences in our implementation underneath VFAT.

6.1 Graceful Degradation

We now present an overview of the basic operation of graceful degradation within Alexander.

6.1.1 The Indirection Map

Similar to any other SCSI-based RAID system, Alexander presents host systems with a linear logical block address space. Internally, Alexander must place blocks so as to facilitate graceful degradation. Thus, to control placement, Alexander introduces a transparent level of indirection between the logical array used by the file system and physical placement onto the disks via the *indirection map (imap)*; similar structures have been used by others [12, 46, 47]. Unlike most of these other systems, this imap only maps every *live* logical file system block to its replica list, *i.e.*, all its physical locations. All *unmapped* blocks are considered free and are candidates for use by D-GRAID.

6.1.2 Reads

Handling block read requests at the D-GRAID level is straightforward. Given the logical address of the block, Alexander looks in the imap to find the replica list and issues the read request to one of its replicas. The choice of which replica to read from can be based on various criteria [47]; currently Alexander uses a randomized selection.

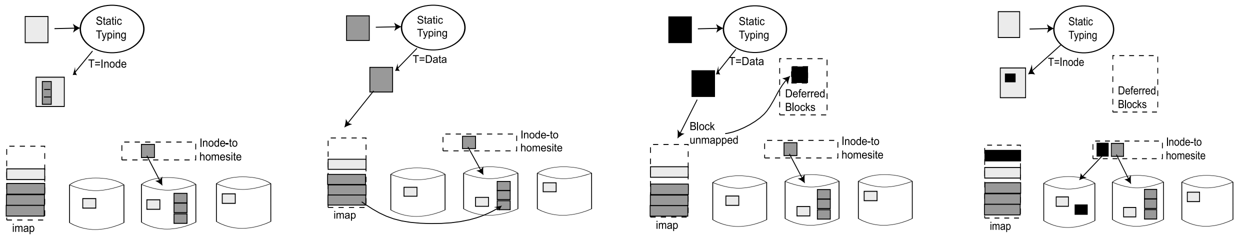

Figure 4: **Anatomy of a Write** *This figure depicts the control flow during a sequence of* `write` *operations in Alexander. In the first figure, an inode block is written; Alexander observes the contents of the inode block and identifies the newly added inode. It then selects a home site for the inode and creates physical mappings for the blocks of the inode, in that home site. Also, the inode block is aggressively replicated. In the next figure, Alexander observes a write to a data block from the same inode; since it is already mapped, the write goes directly to the physical block. In the third figure, Alexander gets a write to an unmapped data block; it therefore defers the block, and when Alexander finally observes the corresponding inode (in the fourth figure), it creates the relevant mappings, observes that one of its blocks is deferred, and therefore issues the deferred write to the relevant home site.*

6.1.3 Writes

In contrast to reads, write requests are more complex to handle. Exactly how Alexander handles the write request depends on the *type* of the block that is written. Figure 4 depicts some common cases.

If the block is a static meta-data block (*e.g.*, an inode or a bitmap block) that is as of yet unmapped, Alexander allocates a physical block in each of the disks where a replica should reside, and writes to all of the copies. Note that Alexander can easily detect static block types such as inode and bitmap blocks underneath many UNIX file systems simply by observing the logical block address.

When an inode block is written, D-GRAID scans the block for newly added inodes; to understand which inodes are new, D-GRAID compares the newly written block with its old copy, a process referred to as block differencing. For every new inode, D-GRAID selects a home site to lay out blocks belonging to the inode, and records it in the *inode-to-homesite* hashtable. This selection of home site is done to balance space allocation across physical disks. Currently, D-GRAID uses a greedy approach; it selects the home site with the least disk space utilization.

If the write is to an unmapped block in the data region (*i.e.*, a data block, an indirect block, or a directory block), the allocation cannot be done until D-GRAID knows which file the block belongs to, and thus, its actual home site. In such a case, D-GRAID places the block in a *deferred block list* and does not write it to disk until it learns which file the block is associated with. Since a crash before the inode write would make the block inaccessible by the file system anyway, the in-memory deferred block list is not a reliability concern.

D-GRAID also looks for newly added block pointers when an inode (or indirect) block is written. If the newly added block pointer refers to an unmapped block, D-GRAID adds a new entry in the imap, mapping the logical block to a physical block in the home site assigned to the corresponding inode. If any newly added pointer refers to a block in the deferred list, D-GRAID removes the block from the deferred list and issues the write to the appropriate physical block(s). Thus, writes are deferred only for blocks that are written *before* the corresponding owner inode blocks. If the inode is written first, subsequent data writes will be already mapped and sent to disk directly.

Another block type of interest that D-GRAID looks for is the data bitmap block. Whenever a data bitmap block is written, D-GRAID scans through it looking for newly freed data blocks. For every such freed block, D-GRAID removes the logical-to-physical mapping if one exists and frees the corresponding physical blocks. Further, if a block that is currently in the deferred list is freed, the block is removed from the deferred list and the write is suppressed; thus, data blocks that are written by the file system but deleted before their corresponding inode is written to disk do not generate extra disk traffic, similar to optimizations found in many file systems [39]. Removing such blocks from the deferred list is important because in the case of freed blocks, Alexander may never observe an owning inode. Thus, every deferred block stays in the deferred list for a bounded amount of time, until either an inode owning the block is written, or a bitmap block indicating deletion of the block is written. The exact duration depends on the delayed write interval of the file system.

6.1.4 Block Reuse

We now discuss a few of the more intricate issues involved with implementing graceful degradation. The first such issue is block reuse. As existing files are deleted or truncated and new files are created, blocks that were once part of one file may be reallocated to some other file. Since D-GRAID needs to place blocks onto the correct home site, this reuse of blocks needs to be detected and acted upon. D-GRAID handles block reuse in the following manner: whenever an inode block or an indirect block is written, D-GRAID examines each valid block pointer to see if its physical block mapping matches the home site allocated for the corresponding inode. If not, D-GRAID changes the mapping for the block to the correct home site. However, it is possible that a write to this block (that was made in the context of the new file) went to the old home site, and hence needs to be copied from its old physical location to the new location. Blocks that must be copied are added to a *pending copies list*; a background thread copies the blocks to their new home site and frees the old physical locations when the copy completes.

6.1.5 Dealing with Imperfection

Another difficulty that arises in semantically-smart disks underneath typical file systems is that exact knowledge of the type of a dynamically-typed block is impossible to obtain, as discussed in Section 5. Thus, Alexander must handle incorrect type classification for data blocks (*i.e.*, file data, directory, and indirect blocks).

For example, D-GRAID must understand the contents of indirect blocks, because it uses the pointers therein to place a file's blocks onto its home site. However, due to lack of perfect knowledge, the fault-isolated placement of a file might be compromised (note that data loss or corruption is not an issue). Our goal in dealing with imperfection is thus to conservatively avoid it when possible, and eventually detect and handle it in all other cases.

Specifically, whenever a block construed to be an indirect block is written, we assume it is a valid indirect block. Thus, for every live pointer in the block, D-GRAID must take some action. There are two cases to consider. In the first case, a pointer could refer to an unmapped logical block. As mentioned before, D-GRAID then creates a new mapping in the home site corresponding to the inode to which the indirect block belongs. If this indirect block (and pointer) is valid, this mapping is the correct mapping. If this indirect block is misclassified (and consequently, the pointer invalid), D-GRAID detects that the block is free when it observes the data bitmap write, at which point the mapping is removed. If the block is allocated to a file before the bitmap is written, D-GRAID detects the reallocation during the inode write corresponding to the new file, creates a new mapping, and copies the data contents to the new home site (as discussed above).

In the second case, a potentially corrupt block pointer could point to an already mapped logical block. As discussed above, this type of block reuse results in a new mapping and copy of the block contents to the new home site. If this indirect block (and hence, the pointer) is valid, this new mapping is the correct one for the block. If instead the indirect block is a misclassification, Alexander wrongly copies over the data to the new home site. Note that the data is still accessible; however, the original file to which the block belongs, now has one of its blocks in the incorrect home site. Fortunately, this situation is transient, because once the inode of the file is written, D-GRAID detects this as a reallocation and creates a new mapping back to the original home site, thereby restoring its correct mapping. Files which are never accessed again are properly laid out by an infrequent sweep of inodes that looks for rare cases of improper layout.

Thus, without any optimizations, D-GRAID will eventually move data to the correct home site, thus preserving graceful degradation. However, to reduce the number of times such a misclassification occurs, Alexander makes an assumption about the contents of indirect blocks, specifically that they contain some number of valid unique pointers, or null pointers. Alexander can leverage this assumption to greatly reduce the number of misclassifications, by performing an integrity check on each supposed indirect block. The integrity check, which is reminiscent of work on conservative garbage collection [5], returns true if all the "pointers" (4-byte words in the block) point to valid data addresses within the volume and all non-null pointers are unique. Clearly, the set of blocks that pass this integrity check could still be corrupt if the data contents happened to exactly evade our conditions. However, a test run across the data blocks of our file system indicates that only a small fraction of data blocks (less than 0.1%) would pass the test; only those blocks that pass the test *and* are reallocated from a file data block to an indirect block would be misclassified.

6.1.6 Access-driven Diffusion

Another issue that D-GRAID must address is performance. Fault-isolated data placement improves availability but at the cost of performance. Data accesses to blocks of a large file, or, with directory-based grouping, to files within the same directory, are no longer parallelized. To improve performance, Alexander performs access-driven diffusion, monitoring block accesses to determine which are "hot", and then "diffusing" those blocks via replication across the disks of the system to enhance parallelism.

Access-driven diffusion can be achieved at both the logical and physical levels of a disk volume. In the logical approach, access to individual files is monitored, and those considered hot are diffused. However, per-file replication fails to capture sequentiality across multiple small files, for example, those in a single directory. Therefore we instead pursue a physical approach, in which Alexander replicates segments of the logical address space across the disks of the volume. Since file systems are good at allocating contiguous logical blocks for a single file, or to files in the same directory, replicating logical segments is likely to identify and exploit most common access patterns.

To implement access-driven diffusion, Alexander divides the logical address space into multiple segments, and during normal operation, gathers various statistics about the utilization and access patterns to each segment. A background thread selects logical segments that are likely to benefit most from access-driven diffusion and diffuses a copy across the drives of the system. Subsequent reads and writes first go to these replicas, with background updates sent to the original blocks. The imap entry for the block indicates which copy is up to date.

The amount of disk space to allocate to performance-oriented replicas presents an important policy decision. The initial policy that Alexander implements is to reserve a certain minimum amount of space (specified by the system administrator) for these replicas, and then opportunistically use the free space available in the array for additional replication. This approach is similar to that used by AutoRAID for mirrored data [47], except that AutoRAID cannot identify data that is considered "dead" by the file system once written; in contrast, D-GRAID can use semantic knowledge to identify which blocks are free.

6.2 Live-block Recovery

To implement live-block recovery, D-GRAID must understand which blocks are live. This knowledge must be correct in that no block that is live is considered dead, as that would lead to data loss. Alexander tracks this information by observing bitmap and data block traffic. Bitmap blocks tell us the liveness state of the file system that has been reflected to disk. However, due to reordering and delayed updates, it is not uncommon to observe a write to a data block whose corresponding bit has not yet been set in the data bitmap. To account for this, D-GRAID maintains a duplicate copy of all bitmap blocks, and whenever it sees a write to a block, sets the corresponding bit in the local copy of the bitmap. The duplicate copy is synchronized with the file system copy when the data bitmap block is written by the file system. This *conservative bitmap table* thus reflects a superset of all live blocks in the file system, and can be used to perform live-block recovery. Note that we assume the pre-allocation state of the bitmap will not be written to disk after a subsequent allocation; the locking in Linux and other modern systems already ensures this. Though this technique guarantees that a live block is never classified as dead, it is possible for the disk to consider a block live far longer than it actually is. This situation would arise, for example, if the file system writes deleted blocks to disk.

To implement live-block recovery, Alexander simply uses the conservative bitmap table to build a list of blocks which need to be restored. Alexander then proceeds through the list and copies all live data onto the hot spare.

6.3 Other Aspects of Alexander

There are a host of other aspects of the implementation that are required for a successful prototype but that we cannot discuss at length due to space limitations. For example, we found that preserving the logical contiguity of the file system was important in block allocation, and thus developed mechanisms to enable such placement. Directory-based grouping also requires more sophistication in the implementation, to handle the further deferral of writes until a parent directory block is written. "Just in time" block allocation prevents misclassified indirect blocks from causing spurious physical block allocation. Deferred list management introduces some tricky issues when there is not enough memory. Alexander also preserves "sync" semantics by not returning success on inode block writes until deferred block writes that were waiting on the inode complete. There are a number of structures that Alexander maintains, such as the imap, that must be reliably committed to disk and preferably, for good performance, buffered in a small amount of non-volatile RAM.

The most important component that is missing from Alexander is the decision on which "popular" (read-only) directories such as `/usr/bin` to replicate widely, and when to do so. Although Alexander contains the proper mechanisms to perform such replication, the policy space remains unexplored. However, our initial experience indicates that a simple approach based on monitoring frequency of inode access time updates may likely be effective. An alternative approach allows administrators to specify directories that should be treated in this manner.

One interesting issue that required a change from our design was the behavior of Linux ext2 under partial disk failure. When a process tries to read a data block that is unavailable, ext2 issues the read and returns an I/O failure to the process. When the block becomes available again (*e.g.*, after recovery) and a process issues a read to it, ext2 will again issue the read, and everything works as expected. However, if a process tries to open a file whose inode is unavailable, ext2 marks the inode as "suspicious" and will never again issue an I/O request to the inode block, even if Alexander has recovered the block. To avoid a change to the file system and retain the ability to recover failed inodes, Alexander replicates inode blocks as it does namespace meta-data, instead of collocating them with the data blocks of a file.

6.4 Alexander the FAT

Overall, we were surprised by the many similarities we found in implementing D-GRAID underneath ext2 and VFAT. For example, VFAT also overloads data blocks, using them as either user data blocks or directories; hence Alexander must defer classification of those blocks in a manner similar to the ext2 implementation.

However, there were a few instances where the VFAT implementation of D-GRAID differed in interesting ways from the ext2 version. For example, the fact that all pointers of a file are located in the file allocation table made a number of aspects of D-GRAID much simpler to implement; in VFAT, there are no indirect pointers to worry about. We also ran across the occasional odd behavior in the Linux implementation of VFAT. For example, Linux would write to disk data blocks that were allocated but then freed, avoiding an obvious and common file system optimization. Although this was more indicative of the untuned nature of the Linux implementation, it served as yet another indicator of how semantic disks must be wary of any assumptions they make about file system behavior.

7 Evaluating Alexander

We now present a performance evaluation of Alexander. We focus primarily on the Linux ext2 variant, but also

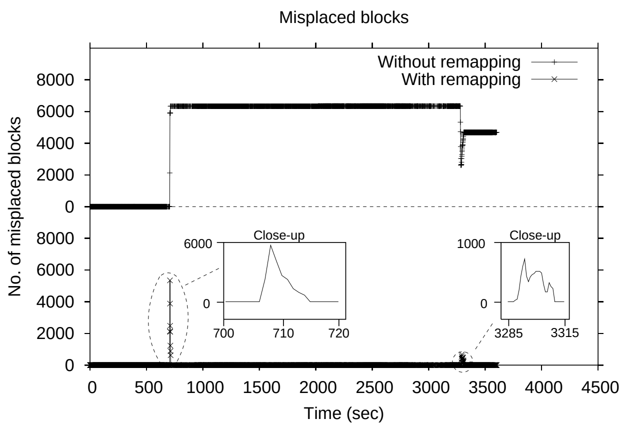

Figure 5: **Errors in Placement.** *The figure plots the number of blocks wrongly laid out by Alexander over time, while running a busy hour of the HP Trace. The experiment was run over 4 disks, and the total number of blocks accessed in the trace was 418000.*

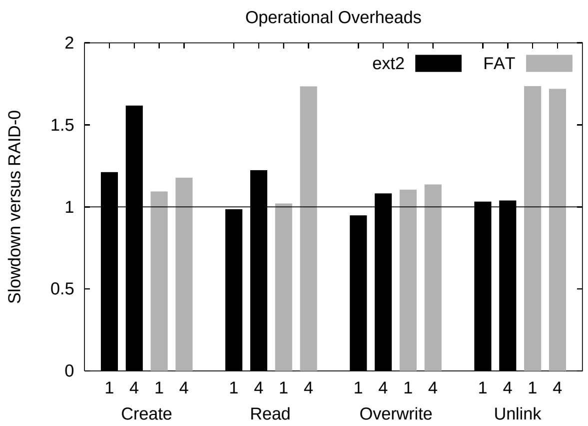

Figure 6: **Time Overheads.** *The figure plots the time overheads observed on D-GRAID Level 0 versus RAID Level 0 across a series of microbenchmarks. The tests are run on 1 and 4 disk systems. In each experiment, 3000 operations were enacted (e.g., 3000 file creations), with each operation on a 64 KB file.*

include some baseline measurements of the VFAT system. We wish to answer the following questions:

- *Does Alexander work correctly?*
- *What time overheads are introduced?*
- *How effective is access-driven diffusion?*
- *How fast is live-block recovery?*
- *What overall benefits can we expect from D-GRAID?*
- *How complex is the implementation?*

7.1 Platform

The Alexander prototype is constructed as a software RAID driver in the Linux 2.2 kernel. File systems mount the pseudo-device and use it as if it were a normal disk. Our environment is excellent for understanding many of the issues that would be involved in the construction of a "real" hardware D-GRAID system; however, it is also limited in the following ways. First, and most importantly, Alexander runs on the same system as the host OS and applications, and thus there is interference due to competition for resources. Second, the performance characteristics of the microprocessor and memory system may be different than what is found within an actual RAID system. In the following experiments, we utilize a 550 MHz Pentium III and four 10K-RPM IBM disks.

Does Alexander work correctly? Alexander is more complex than simple RAID systems. To ensure that Alexander operates correctly, we have put the system through numerous stress tests, moving large amounts of data in and out of the system without problems. We have also extensively tested the corner cases of the system, pushing it into situations that are difficult to handle and making sure that the system degrades gracefully and recovers as expected. For example, we repeatedly crafted microbenchmarks to stress the mechanisms for detecting block reuse and for handling imperfect information about dynamically-typed blocks. We have also constructed benchmarks that write user data blocks to disk that contain "worst case" data, *i.e.*, data that appears to be valid directory entries or indirect pointers. In all cases, Alexander was able to detect which blocks were indirect blocks and move files and directories into their proper fault-isolated locations.

To verify that Alexander places blocks on the appropriate disk, we instrumented the file system to log block allocations. In addition, Alexander logs events of interest such as assignment of a home site for an inode, creation of a new mapping for a logical block, re-mapping of blocks to a different homesite and receipt of logical writes from the file system. To evaluate the behavior of Alexander on a certain workload, we run the workload on Alexander, and obtain the time-ordered log of events that occurred at the file system and Alexander. We then process this log off-line and look for the number of blocks wrongly laid out at any given time.

We ran this test on a few hours of the HP Traces, and found that in many of the hours we examined, the number of blocks that were misplaced even temporarily was quite low, often less than 10 blocks. We report detailed results for one such hour of the trace where we observed the greatest number of misplaced blocks, among the hours we examined. Figure 5 shows the results.

The figure has two parts. The bottom part shows the normal operation of Alexander, with the capability to react to block reuse by remapping (and copying over) blocks to the correct homesite. As the figure shows, Alexander is able to quickly detect wrongly placed blocks and remap them appropriately. Further, the number of such blocks misplaced temporarily is only about 1% of the total number of blocks accessed in the trace. The top part of the figure shows the number of misplaced blocks for the same experiment, but assuming that the remapping did not occur. As can be expected, those delinquent blocks remain misplaced. The dip towards the end of the trace occurs

	Run-time (seconds)	Blocks Written Total	Meta data	Unique
RAID-0	69.25	101297	–	–
D-GRAID$_1$	61.57	93981	5962	1599
D-GRAID$_2$	66.50	99416	9954	3198
D-GRAID$_3$	73.50	101559	16976	4797
D-GRAID$_4$	78.79	113222	23646	6396

Table 2: **Performance on postmark.** *The table compares the performance of D-GRAID Level 0 with RAID-0 on the Postmark benchmark. Each row marked D-GRAID indicates a specific level of metadata replication. The first column reports the benchmark run-time and the second column shows the number of disk writes incurred. The third column shows the number of disk writes that were to metadata blocks, and the fourth column indicates the number of unique metadata blocks that are written. The experiment was run over 4 disks.*

because some of the misplaced blocks are later assigned to a file in that homesite itself, accidentally correcting the original misplacement.

What time overheads are introduced? We now explore the time overheads that arise due to semantic inference. This primarily occurs when new blocks are written to the file system, such as during file creation. Figure 6 shows the performance of Alexander under a simple microbenchmark. As can be seen, allocating writes are slower due to the extra CPU cost involved in tracking fault-isolated placement. Reads and overwrites perform comparably with RAID-0. The high unlink times of D-GRAID on FAT is because FAT writes out data pertaining to deleted files, which have to be processed by D-GRAID as if it were newly allocated data. Given that the implementation is untuned and the infrastructure suffers from CPU and memory contention with the host, we believe that these are worst case estimates of the overheads.

Another cost of D-GRAID that we explore is the overhead of metadata replication. For this purpose, we choose Postmark [25], a metadata intensive file system benchmark. We slightly modified Postmark to perform a `sync` before the deletion phase, so that all metadata writes are accounted for, making it a pessimistic evaluation of the costs. Table 2 shows the performance of Alexander under various degrees of metadata replication. As can be seen from the table, synchronous replication of metadata blocks has a significant effect on performance for metadata intensive workloads (the file sizes in Postmark range from 512 bytes to 10 KB). Note that Alexander performs better than default RAID-0 for lower degrees of replication because of better physical block allocation; since ext2 looks for a contiguous free chunk of 8 blocks to allocate a new file, its layout is sub-optimal for small files.

The table also shows the number of disk writes incurred during the course of the benchmark. The percentage of extra disk writes roughly accounts for the difference in performance between different replication levels, and these extra writes are mostly to metadata blocks. However, when we count the number of unique physical writes to metadata blocks, the absolute difference between different replication levels is small. This suggests that lazy propagation of updates to metadata block replicas, perhaps during idle time or using freeblock scheduling, can greatly reduce the performance difference, at the cost of added complexity. For example, with lazy update propagation (*i.e.*, if the replicas were updated only once), D-GRAID$_4$ would incur only about 4% extra disk writes.

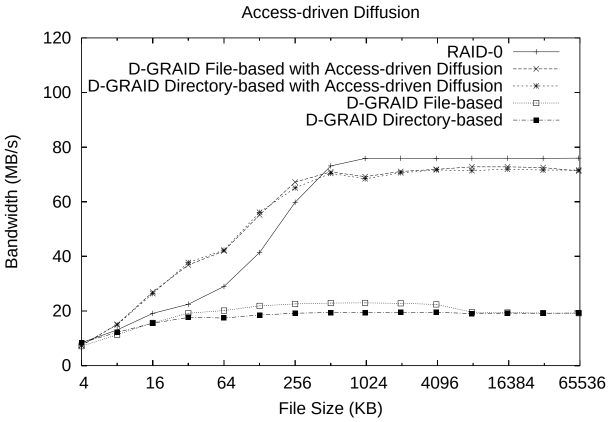

Figure 7: **Access-driven Diffusion.** *The figure presents the performance of D-GRAID Level 0 and standard RAID-0 under a sequential workload. In each experiment, a number of files of size x are read sequentially, with the total volume of data fixed at 64 MB. D-GRAID performs better for smaller files due to better physical block layout.*

We also played back a portion of the HP traces for 20 minutes against a standard RAID-0 system and D-GRAID over four disks. The playback engine issues requests at the times specified in the trace, with an optional speedup factor; a speedup of 2× implies the idle time between requests was reduced by a factor of two. With speedup factors of 1× and 2×, D-GRAID delivered the same per-second operation throughput as RAID-0, utilizing idle time in the trace to hide its extra CPU overhead. However, with a scaling factor of 3×, the operation throughput lagged slightly behind, with D-GRAID showing a slowdown of up to 19.2% during the first one-third of the trace execution, after which it caught up due to idle time.

How effective is access-driven diffusion? We now show the benefits of access-driven diffusion. In each trial of this experiment, we perform a set of sequential file reads, over files of increasing size. We compare standard RAID-0 striping to D-GRAID with and without access-driven diffusion. Figure 7 shows the results of the experiment.

As we can see from the figure, without access-driven diffusion, sequential access to larger files run at the rate of a single disk in the system, and thus do not benefit from the potential parallelism. With access-driven diffusion, performance is much improved, as reads are directed

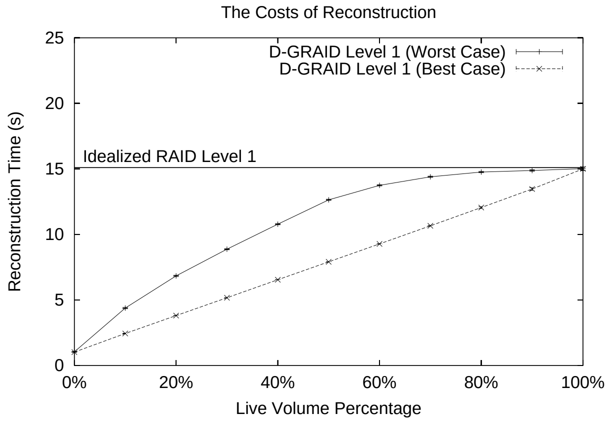

Figure 8: **Live-block Recovery.** *The figure shows the time to recover a failed disk onto a hot spare in a D-GRAID Level 1 (mirrored) system using live-block recovery. Two lines for D-GRAID are plotted: in the worst case, live data is spread across the entire 300 MB volume, whereas in the best case it is compacted into the smallest contiguous space possible. Also plotted is the recovery time of an idealized RAID Level 1.*

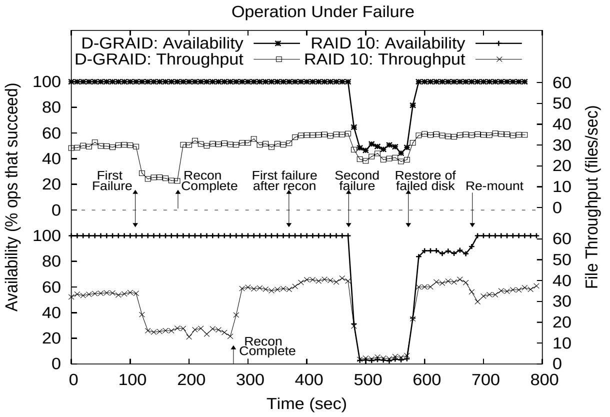

Figure 9: **Availability Profile.** *The figure shows the operation of D-GRAID Level 1 and RAID 10 under failures. The 3 GB array consists of 4 data disks and 1 hot spare. After the first failure, data is reconstructed onto the hot spare, D-GRAID recovering much faster than RAID 10. When two more failures occur, RAID 10 loses almost all files, while D-GRAID continues to serve 50% of its files. The workload consists of read-modify-writes of 32 KB files randomly picked from a 1.3 GB working set.*

to the diffused copies across all of the disks in the system. Note that in the latter case, we arrange for the files to be already diffused before the start of the experiment, by reading them a certain threshold number of times. Investigating more sophisticated policies for when to initiate access-driven diffusion is left for future work.

How fast is live-block recovery? We now explore the potential improvement seen with live-block recovery. Figure 8 presents the recovery time of D-GRAID while varying the amount of live file system data.

The figure plots two lines: worst case and best case live-block recovery. In the worst case, live data is spread throughout the disk, whereas in the best case it is compacted into a single portion of the volume. From the graph, we can see that live-block recovery is successful in reducing recovery time, particularly when a disk is less than half full. Note also the difference between worst case and best case times; the difference suggests that periodic disk reorganization [41] could be used to speed recovery, by moving all live data to a localized portion.

What overall benefits can we expect from D-GRAID? We next demonstrate the improved availability of Alexander under failures. Figure 9 shows the availability and performance observed by a process randomly accessing whole 32 KB files, running above D-GRAID and RAID-10. To ensure a fair comparison, both D-GRAID and RAID-10 limit their reconstruction rate to 10 MB/s.

As the figure shows, reconstruction of the 3 GB volume with 1.3 GB live data completes much faster (68 s) in D-GRAID compared to RAID-10 (160 s). Also, when the extra second failure occurs, the availability of RAID-10 drops to near zero, while D-GRAID continues with about 50 % availability. Surprisingly, after restore, RAID-10 still fails on certain files; this is because Linux does not retry inode blocks once they fail. A remount is required before RAID-10 returns to full availability.

How complex is the implementation? We briefly quantify the implementation complexity of Alexander. Table 3 shows the number of C statements required to implement the different components of Alexander. From the table, we can see that the core file system inferencing module for ext2 requires only about 1200 lines of code (counted with number of semicolons), and the core mechanisms of D-GRAID contribute to about 2000 lines of code. The rest is spent on a hash table, AVL tree and wrappers for memory management. Compared to the tens of 1000's of lines of code already comprising modern array firmware, we believe that the added complexity of D-GRAID is not that significant.

8 Discussion

In this section, we first compare our semantic-disk based approach to alternative methods of implementing D-GRAID, and then discuss some possible concerns about the commercial feasibility of such semantic disk systems.

8.1 Alternative Approaches

Our semantic disk based approach is one of few different ways of implementing D-GRAID, each with its own trade-offs. Similar to modern processors that innovate beneath unchanged instruction sets, a semantic disk level implementation facilitates ease of deployment and inter-operability with unchanged client infrastructure, perhaps making it more pragmatic. The cost of this approach, however, is the complexity in rediscovering semantic knowledge and being tolerant to inaccuracies.

An alternative approach is to change the interface between file systems and storage, to convey richer information across both layers. For instance, the storage system could expose failure boundaries to the file system [9], and

	Semicolons	Total
D-GRAID Generic		
Setup + fault-isolated placement	1726	3557
Physical block allocation	322	678
Access driven diffusion	108	238
Mirroring + live block recovery	248	511
Internal memory management	182	406
Hashtable/Avl tree	724	1706
File System Specific		
SDS Inferencing: ext2	1252	2836
SDS Inferencing: VFAT	630	1132
Total	5192	11604

Table 3: **Code size for Alexander implementation.** *The number of lines of code needed to implement Alexander is shown. The first column shows the number of semicolons and the second column shows the total number of lines, including white-spaces and comments.*

then the file system could explicitly allocate blocks in a fault-isolated manner. Alternatively, the file system could tag each write with a logical fault-container ID, which can then be used by the storage system to implement fault-isolated data placement. These techniques, while being intrusive on existing infrastructure and software base, are conceivably less complex than our approach.

Object-based storage [16] is one such new interface being considered, which makes the file boundaries more visible at the storage layer. However, even with an object-based interface, semantically-smart technology might still be relevant to discover relationships across objects; for instance inferring that a directory object points to a set of file objects which need to collocated.

8.2 Commercial Feasibility

By definition, D-GRAID and other semantically-smart storage systems have more detailed knowledge of the file system that is using them. Embedding a higher degree of functionality within the storage system leads to some concerns on the commercial feasibility of such systems.

The first concern that arises is that placing semantic knowledge within the disk system ties the disk system too intimately to the file system above. For example, if the file system's on-disk structure changes, the storage system may have to change as well. We believe this issue is not likely to be problematic. On-disk formats evolve slowly, for reasons of backwards compatibility. For example, the basic structure of FFS-based file systems has not changed since its introduction in 1984, a period of almost twenty years [28]; the Linux ext2 file system, introduced in roughly 1994, has had the exact same layout for its lifetime. finally, the ext3 journaling file system [45] is backwards compatible with ext2 on-disk layout and the new extensions to the FreeBSD file system [10] are backwards compatible as well. We also have evidence that storage vendors are already willing to maintain and support software specific to a file system; for example, the EMC Symmetrix storage system [11] comes with software that can understand the format of most common file systems.

The second concern is that the storage system needs semantic knowledge for each file system with which it interacts. Fortunately, there are not a large number of file systems that would need to be supported to cover a large fraction of the usage population. If such a semantic storage system is used with a file system that it does not support, the storage system could detect that the file system does not conform to its expectations and turn off its special functionality (*e.g.*, in the case of D-GRAID, revert to normal RAID layout). Such detection can be done by simple techniques such as observing the file system identifier in the partition table.

One final concern that arises is that too much processing will be required within the disk system. We do not believe this to be a major issue, because of the general trend of increasing disk system intelligence [1, 37]; as processing power increases, disk systems are likely to contain substantial computational abilities. Indeed, modern storage arrays already exhibit the fruits of Moore's Law; for example, the EMC Symmetrix storage server can be configured with up to 80 processors and 64 GB of RAM [11].

9 Related Work

D-GRAID draws on related work from a number of different areas, including distributed file systems and traditional RAID systems. We discuss each in turn.

Distributed File Systems: Designers of distributed file systems have long ago realized the problems that arise when spreading a directory tree across different machines in a system. For example, Walker *et al.* discuss the importance of directory namespace replication within the Locus distributed system [35]. The Coda mobile file system also takes explicit care with regard to the directory tree [27]. Specifically, if a file is cached, Coda makes sure to cache every directory up to the root of the directory tree. By doing so, Coda can guarantee that a file remains accessible should a disconnection occur. Perhaps an interesting extension to our work would be to reconsider host-based in-memory caching with availability in mind. Also, Slice [3] tries to route namespace operations for all files in a directory to the same server.

More recently, work in wide-area file systems has also re-emphasized the importance of the directory tree. For example, the Pangaea file system aggressively replicates the entire tree up to the root on a node when a file is accessed [42]. The Island-based file system also points out the need for "fault isolation" but in the context of wide-area storage systems; their "one island principle" is quite similar to fault-isolated placement in D-GRAID [24].

Finally, p2p systems such as PAST that place an entire file on a single machine have similar load balancing issues [40]. However, the problem is more difficult in the p2p space due to the constraints of file placement; block migration is much simpler in a centralized storage array.

Traditional RAID Systems: We also draw on the long history of research in classic RAID systems. From AutoRAID [47] we learned both that complex functionality could be embedded within a modern storage array, and that background activity could be utilized successfully in such an environment. From AFRAID [43], we learned that there could be a flexible trade-off between performance and reliability, and the value of delaying updates.

Much of RAID research has focused on different redundancy schemes. While early work stressed the ability to tolerate single-disk failures [4, 32, 33], later research introduced the notion of tolerating multiple-disk failures within an array [2, 6]. We stress that our work is complementary to this line of research; traditional techniques can be used to ensure full file system availability up to a certain number of failures, and D-GRAID techniques ensure graceful degradation under additional failures. A related approach is parity striping [18] which stripes only the parity and not data; while this would achieve some fault isolation, the layout is still oblivious of the semantics of the data; blocks will have the same level of redundancy irrespective of their importance (*i.e.*, meta-data vs data), so multiple failures could still make the entire file system inaccessible. A number of earlier works also emphasize the importance of hot sparing to speed recovery time in RAID arrays [21, 29, 32]. Our work on semantic recovery is also complementary to those approaches.

Finally, note that term "graceful degradation" is sometimes used to refer to the performance characteristics of redundant disk systems under failure [22, 36]. This type of graceful degradation is different from what we discuss in this paper; indeed, none of those systems continues operation when an unexpected number of failures occurs.

10 Conclusions

> "A robust system is one that continues to operate (nearly) correctly in the presence of some class of errors" *Robert Hagmann [20]*

D-GRAID turns the simple binary failure model found in most storage systems into a continuum, increasing the availability of storage by continuing operation under partial failure and quickly restoring live data after a failure does occur. In this paper, we have shown the potential benefits of D-GRAID, established the limits of semantic knowledge, and have shown how a successful D-GRAID implementation can be achieved despite these limits. Through simulation and the evaluation of a prototype implementation, we have found that D-GRAID can be built underneath a standard block-based interface, without any file system modification, and that it delivers graceful degradation and live-block recovery, and, through access-driven diffusion, good performance.

We conclude with a discussions of the lessons we have learned in the process of implementing D-GRAID:

- **Limited knowledge within the disk does not imply limited functionality.** One of the main contributions of this paper is a demonstration of both the limits of semantic knowledge, as well as the "proof" via implementation that despite such limitations, interesting functionality can be built inside of a semantically-smart disk system. We believe any semantic disk system must be careful in its assumptions about file system behavior, and hope that our work can guide others who pursue a similar course.
- **Semantically-smart disks would be easier to build with some help from above.** Because of the way file systems reorder, delay, and hide operations from disks, reverse engineering exactly what they are doing at the SCSI level is difficult. We believe that small modifications to file systems could substantially lessen this difficulty. For example, if the file system could inform the disk whenever it believes the file system structures are in a consistent on-disk state, many of the challenges in the disk would be lessened. This is one example of many small alterations that could ease the burden of semantic disk development.
- **Semantically-smart disks stress file systems in unexpected ways.** File systems were not built to operate on top of disks that behave as D-GRAID does; specifically, they may not behave particularly well when part of a volume address space becomes unavailable. Perhaps because of its heritage as an OS for inexpensive hardware, Linux file systems handle unexpected conditions fairly well. However, the exact model for dealing with failure is inconsistent: data blocks could be missing and then reappear, but the same is not true for inodes. As semantically-smart disks push new functionality into storage, file systems would likely have to evolve to accommodate them.
- **Detailed traces of workload behavior are invaluable.** Because of the excellent level of detail available in the HP traces [38], we were able to simulate and analyze the potential of D-GRAID under realistic settings. Many other traces do not contain per-process information, or anonymize file references to the extent that pathnames are not included in the trace, and thus we could not utilize them in our study. One remaining challenge for tracing is to include user data blocks, as semantically-smart disks may be sensitive to the contents. However, the privacy concerns that such a campaign would encounter may be too difficult to overcome.

Acknowledgments

We would like to thank Anurag Acharya, Erik Riedel, Yasushi Saito, John Bent, Nathan Burnett, Timothy Denehy, Brian Forney, Florentina Popovici, and Lakshmi Bairavasundaram for their insightful comments on earlier drafts of the paper. We also would like to thank Richard Golding for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions, many of which have greatly improved the content of this paper. Finally, we thank the Computer Systems Lab for providing a terrific environment for computer science research.

This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, ITR-0325267, IBM, EMC, and the Wisconsin Alumni Research Foundation.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks: programming model, algorithms and evaluation. In *ASPLOS VIII*, San Jose, CA, October 1998.

[2] G. A. Alvarez, W. A. Burkhard, and F. Cristian. Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering. In *ISCA '97*, pages 62–72, 1997.

[3] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing for Scalable Network Storage. *ACM Transactions on Computer Systems*, 20(1), February 2002.

[4] D. Bitton and J. Gray. Disk shadowing. In *VLDB 14*, pages 331–338, Los Angeles, CA, August 1988.

[5] H. Boehm and M. Weiser. Garbage Collection in an Uncooperative Environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[6] W. Burkhard and J. Menon. Disk Array Storage System Reliability. In *FTCS-23*, pages 432–441, Toulouse, France, June 1993.

[7] J. Chapin, M. Rosenblum, S. Devine, T. Lahiri, D. Teodosiu, and A. Gupta. Hive: Fault Containment for Shared-Memory Multiprocessors. In *SOSP '95*, December 1995.

[8] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, June 1994.

[9] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *USENIX '02*, June 2002.

[10] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *FREENIX '02*, Monterey, CA, June 2002.

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

[12] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *USENIX Winter '92*, January 1992.

[13] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. Technical Report CMU-CS-01-166, Carnegie Mellon University, December 2001.

[14] G. R. Ganger, M. K. McKusick, C. A. Soules, and Y. N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM TOCS*, 18(2), May 2000.

[15] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement. In *HICSS '93*, 1993.

[16] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *ASPLOS VIII*, October 1998.

[17] J. Gray. Why Do Computers Stop and What Can We Do About It? In *6th International Conference on Reliability and Distributed Databases*, June 1987.

[18] J. Gray, B. Horst, and M. Walker. Parity Striping of Disc Arrays: Low-cost Reliable Storage with Acceptable Throughput. In *Proceedings of the 16th International Conference on Very Large Data Bases (VLDB 16)*, pages 148–159, Brisbane, Australia, August 1990.

[19] S. D. Gribble. Robustness in Complex Systems. In *HotOS VIII*, Schloss Elmau, Germany, May 2001.

[20] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *SOSP '87*, November 1987.

[21] M. Holland, G. Gibson, and D. Siewiorek. Fast, on-line failure recovery in redundant disk arrays. In *FTCS-23*, France, 1993.

[22] H.-I. Hsiao and D. DeWitt. Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines. In *6th International Data Engineering Conference*, 1990.

[23] IBM. ServeRAID - Recovering from multiple disk failures. http://www.pc.ibm.com/qtechinfo/MIGR-39144.html, 2001.

[24] M. Ji, E. Felten, R. Wang, and J. P. Singh. Archipelago: An Island-Based File System For Highly Available And Scalable Internet Services. In *4th USENIX Windows Symposium*, August 2000.

[25] J. Katcher. PostMark: A New File System Benchmark. Technical Report TR-3022, Network Appliance Inc., Oct 1997.

[26] K. Keeton and J. Wilkes. Automating data dependability. In *Proceedings of the 10th ACM-SIGOPS European Workshop*, pages 93–100, Saint-Emilion, France, September 2002.

[27] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM TOCS*, 10(1), February 1992.

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

[29] J. Menon and D. Mattson. Comparison of Sparing Alternatives for Disk Arrays. In *ISCA '92*, Gold Coast, Australia, May 1992.

[30] Microsoft Corporation. http://www.microsoft.com/hwdev/, December 2000.

[31] C. U. Orji and J. A. Solworth. Doubly Distorted Mirrors. In *SIGMOD '93*, Washington, DC, May 1993.

[32] A. Park and K. Balasubramanian. Providing fault tolerance in parallel secondary storage systems. Technical Report CS-TR-057-86, Princeton, November 1986.

[33] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *SIGMOD '88*, June 1988.

[34] D. A. Patterson. Availability and Maintainability $>>$ Performance: New Focus for a New Century. Key Note at FAST '02, January 2002.

[35] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *SOSP '81*, December 1981.

[36] A. L. N. Reddy and P. Banerjee. Gracefully Degradable Disk Arrays. In *FTCS-21*, pages 401–408, Montreal, Canada, June 1991.

[37] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *Proceedings of the 24th International Conference on Very Large Databases (VLDB 24)*, New York, New York, August 1998.

[38] E. Riedel, M. Kallahalla, and R. Swaminathan. A Framework for Evaluating Storage System Security. In *FAST '02*, pages 14–29, Monterey, CA, January 2002.

[39] 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.

[40] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *SOSP '01*, Banff, Canada, October 2001.

[41] C. Ruemmler and J. Wilkes. Disk Shuffling. Technical Report HPL-91-156, Hewlett Packard Laboratories, 1991.

[42] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *OSDI '02*, Boston, MA, December 2002.

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

[44] M. Sivathanu, V. Prabhakaran, F. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *FAST '03*, San Francisco, CA, March 2003.

[45] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.

[46] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *OSDI '99*, New Orleans, LA, February 1999.

[47] 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.

[48] J. L. Wolf. The Placement Optimization Problem: A Practical Solution to the Disk File Assignment Problem. In *SIGMETRICS '89*, pages 1–10, Berkeley, CA, May 1989.