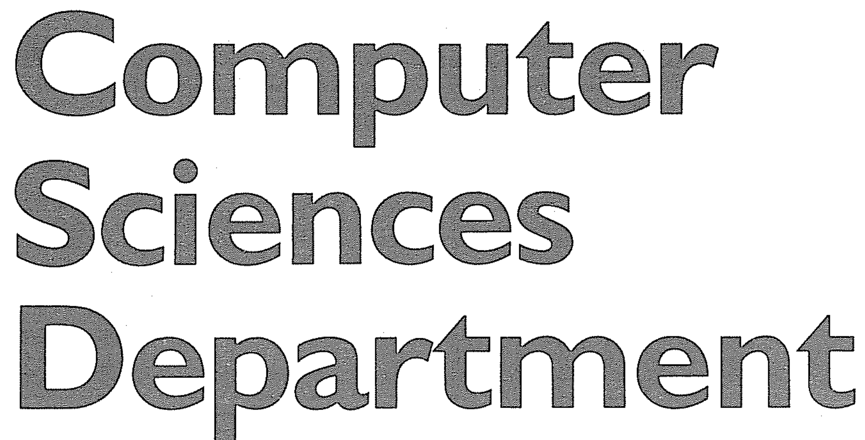

Improving Storage System Availability with D-GRAID

Muthian Sivathanu
Vajayan Prabhakaran
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report #1473

April 2003

UNIVERSITY OF
WISCONSIN
MADISON

Improving Storage System Availability with D-GRAID

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, 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 a storage array is down, entire systems grind to a halt. Downtime is expensive in any setting where computing is at the core of operations; in the on-line business world, for example, millions of dollars per hour are lost when systems are down [20, 28].

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): $\frac{MTBF}{MTBF+MTTR}$ [11]. 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 [2, 4, 6, 12, 17, 25, 26, 27, 36, 40]. By keeping multiple copies of blocks, or through more sophisticated redundancy schemes such as parity-encoding, storage systems can be engineered to tolerate a (small) fixed number of faults. To decrease the time to recovery, "hot spares" can be employed [16, 23, 26, 30]; when a failure occurs, a spare disk is activated and filled with reconstructed data, thus moving the system back into a normal operating mode relatively quickly.

However, in traditional systems, both failure avoidance and recovery techniques are limited, due to the narrow interface between file systems and storage [8]. For example, in a RAID-5 storage array, if one disk too many fails before another is repaired, the entire disk system is corrupted. The reason for this "availability cliff" is that the file system treats the storage array as a single large disk, and therefore obliviously spreads pointers to blocks and other meta-data across all of the disks. When that extra disk fails, recovering a semantically meaningful data unit, such as a file or perhaps some portion of the directory tree, is difficult if not impossible. To restore the system, expensive and error-prone recovery from backup storage is required. Until completed, the entire array remains unavailable, even though most of its disks are still operational.

Further, in a typical storage array, the recovery process must restore all blocks from a failed disk, whether or not they are live in the file system, hence slowing recovery time. In a storage array that is not highly utilized (*e.g.*, it has been configured for performance and not capacity), this is particularly onerous, because only a small fraction of the disks are utilized. To summarize, because of the lack of file system knowledge within the array, many opportunities to improve both the MTBF and MTTR of storage arrays have not been realized.

An ideal storage array fails gracefully. For example, if one-third of the disks of the system are down, only one-third of the data should be unavailable. An ideal storage array recovers more intelligently, restoring live data. In both cases, an ideal storage array ensures that more "important" data is less likely to disappear under failure, and that such data is restored earlier in the recovery process if it does become unavailable. 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 really 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 [37] 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 [5]. Thus, when an unexpected "double" failure occurs [11], 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 [37]. Built underneath of a narrow block-based SCSI storage interface, such a disk system understands file system data structures, including the superblock, allocation bitmaps, inodes, directories, and other important struc-

tures; 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 of both the Linux ext2 and FAT file systems. In this paper, 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 of 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 [10]. 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 demonstrate how our prototype Alexander behaves under microbenchmarks and trace-driven workloads. We find that the construction of D-GRAID is feasible; even with imperfect semantic knowledge, a powerful set of 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 a high level of performance.

The rest of this paper is structured as follows. In Section 2, we present an extended motivation for graceful degradation. Then, in Section 3, we discuss the design principles that underly D-GRAID. We present trace analysis and simulations in Section 4, demonstrating the potential of graceful degradation. In Section 5, we discuss semantic knowledge and its limitations in the context of D-GRAID, and in Section 6, we present the Alexander prototype implementation. In Section 7, we present a detailed evaluation of our prototype, discuss related work in Section 8, and conclude in Section 9.

2 The Case for Graceful Degradation

RAID redundancy techniques typically export a simple failure model. If D or fewer disks fail, the RAID will continue to operate correctly, but perhaps with degraded performance. If more than D disks fail, the RAID becomes entirely unavailable until the problem is corrected, perhaps via a restore from tape. In most RAID schemes, D is small (often 1); thus even though most of the disks are working, the user observes a "failed" disk system.

With graceful degradation, we believe that a RAID system should be able to absolutely tolerate some fixed number of faults (as before), but that excess failures should not be catastrophic; most of the data (an amount proportional to the number of disks still available in the system) should continue 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 that may arise is whether it is realistic to expect a catastrophic failure scenario within a RAID system. For example, in a RAID-5 system, although one disk might fail, it seems highly unlikely that another disk will fail before the one is repaired, especially given the high MTBF's reported by disk manufacturers. We believe these failure scenarios do occur, for two primary reasons. First, correlated faults are often more common in systems than expected [13]. 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 [6]; 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 [11]. A large percentage of human failures occur during mainte-

nance, where (in Gray's words) "the maintenance person typed the wrong command or unplugged the wrong module, thereby introducing a double failure" (page 6) [11].

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 [18]. 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.

One might think that the best approach to dealing with multiple failures would be to employ a higher level of redundancy [4], 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 slow (*e.g.*, updates 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.

3 Design: D-GRAID Expectations

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

3.1 File System Background

Semantic knowledge is obviously highly system specific. In this paper, we discuss the D-GRAID design and implementation in light of two widely differing file systems: Linux ext2 [38] and the Microsoft FAT [24] file system. Inclusion of the FAT file system represents a significant contribution as compared to previous research, which operated solely on UNIX file systems.

The ext2 file system is an intellectual descendant of the Berkeley Fast File System (FFS) [22]. 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. The allocation strategy for files is also quite similar to FFS, in that ext2 tries to group "related" files and their inodes into the same block group; files in the same directory are considered related. Most information about a file, including size and block pointers, are found in the file's inode.

The FAT 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. FAT 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 FAT 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 more information than a typical UNIX directory, including size, permission, and type information.

3.2 Graceful Degradation

To ensure partial availability of data under multiple failures in a RAID array, D-GRAID needs to employ two main techniques. The first is a *fault-isolated data placement* strategy, in which D-GRAID should place 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, the goal of fault-isolated data placement is to ensure that only files that have that disk 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. By widely replicating these meta-data structures, D-GRAID can ensure that all live user 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 is dependent 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.

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-

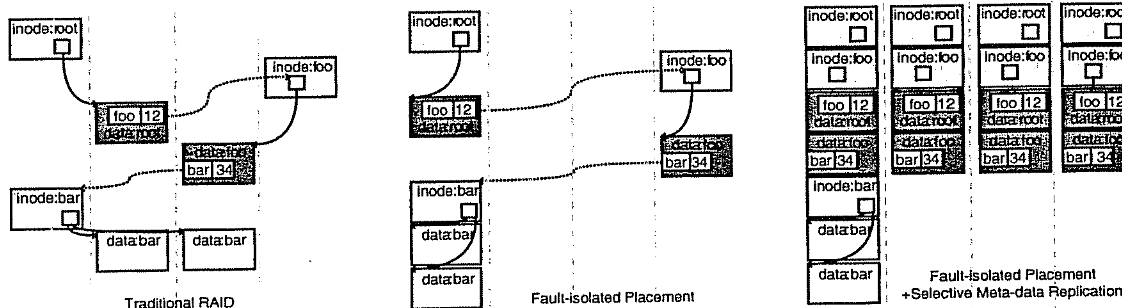

Figure 1: **A Comparison of Layout Schemes.** *These figures depict different layouts of a file "/foo/bar" in a* UNIX *file system starting at the root inode and following down the directory tree to the file data. The example assumes no data redundancy for user file data for simplicity. On the left is 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 inodes and directory blocks of elements high up in the tree, 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.*

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 striping, mirroring, or RAID-5 parity encoding, 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 which must be addressed within D-GRAID. 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 to consider a single file (including its data blocks, inode, and indirect pointers) as the logical unit of data; with this *file-based* grouping technique, however, a user may find that some files in a given directory become unavailable while others do not, a situation that is likely to cause frustration and confusion. We thus must consider alternatives that 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 data placement, instead of placing the blocks of a file across many disks of a system, we instead isolate the blocks to 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 amount of 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 either the foreground (when data is first allocated) or through background migration or both. [1]

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 when compared to sophisticated file placement algorithms [10, 41]. Thus, D-GRAID should make 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 is used to increase 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 an excessive number of 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 implies more resiliency. One difference between traditional RAID and D-GRAID is that the amount

[1] A corner case arises when a file is bigger than the available space on any single disk and cannot be placed in a fault-isolated manner. This problem can be addressed in D-GRAID either through a (expensive) data reorganization or by reserving large extents of free space on a subset of drives in order to be prepared for large file allocations. However, due to the ever-increasing size of disks [14] and the increasing amount of free space found on drives [1], this may not be a first-order concern.

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 update operations. However, others have observed that there is a lack of update activity at higher levels in the directory tree [29], and lazy update propagation can be employed to reduce costs [36].

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 straight-forward 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 straight-forward 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 [21].

4 Exploring Graceful Degradation

We now use simulation and trace analysis in order to evaluate the potential effectiveness of graceful degradation and the impact of different semantic grouping techniques. The simulations use file system traces collected from HP Labs [31], and cover 10 days of activity. In total, there are 250 GB of data spread across 18 logical volumes.

4.1 Space Overheads

We first examine the space overheads that are typical with D-GRAID-style redundancy, specifically due to selective meta-data replication. Table 1 presents the space overheads as measured across all volumes of the HP trace data, for both the Linux ext2 and FAT file systems. We present the data in the most pessimistic manner possible, that is, assuming no replication of user data, and showing the cost of selective meta-data replication as a percentage overhead. Assuming that user data is mirrored, for example, would cut the overheads in half.

As we can see from the table, selective meta-data replication induces only a mild space overhead even under extremely high levels of meta-data redundancy. Even with 16-way redundancy of meta-data, only an extra 8% of space is required in the worst case (FAT-32 with 1 KB blocks). One interesting item to note is that with increasing block size, ext2 uses more space (as expected), but with FAT the overheads actually decrease. This phenomenon is due to the structure of FAT; as block size grows, the file allocation table itself shrinks, although the blocks that contain directory data grow.

	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%
FAT_{1KB}	0.52%	2.07%	8.29%
FAT_{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 FAT, with block size set to 1 KB or 4 KB.*

4.2 Static Availability

We next examine how D-GRAID availability degrades under failure. Figure 2 presents the percent of directories available under two different semantic grouping strategies; a directory is considered available if all of its files are accessible (although subdirectories and their files may not be). The first strategy is file-based grouping, which keeps the information associated with a single file within a failure boundary (*i.e.*, a disk), and 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 system, 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).

From the figure, we observe that graceful degradation works quite well, with the amount of data available proportional to the number of working disks (indeed, availability sometimes degrades in a manner that is better 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 in practice to very good data availability under failure. Compare this to the worst possible behavior under failure, in which a few disk crashes lead to complete data unavailability. We also can conclude that if file-based grouping is used, it is likely that some files in a directory will "disappear" under failure, perhaps leading to user dissatisfaction.

4.3 Dynamic Availability

While static data availability under failure is a good indicator that our approach for graceful degradation will work as expected, it fails to answer a more fundamental question: how often will users or applications be oblivious that the D-GRAID is operating in degraded mode? To answer this question, we present a simulation of dynamic

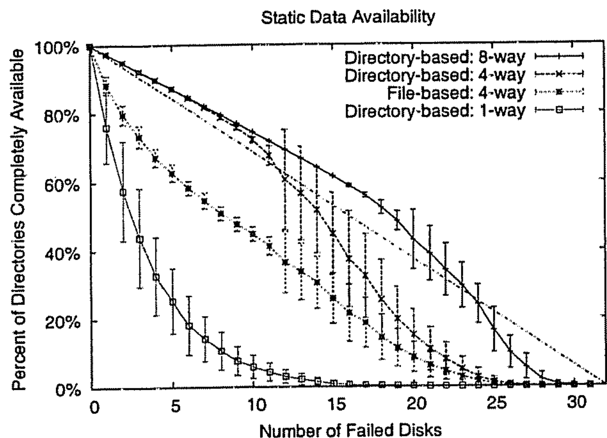

Figure 2: **Static Data Availability.** *The percent of entire directories available is shown under increasing disk failures. The simulated system consists of 32 disks, and is loaded with the 250 GB from the HP trace. Two different strategies for semantic grouping are shown: file-based and directory-based. 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.*

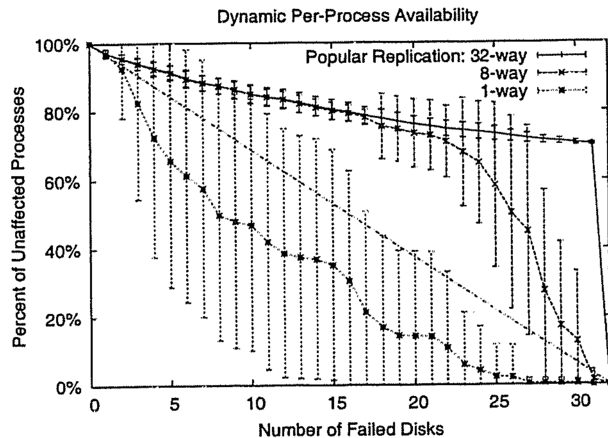

Figure 3: **Dynamic Data Availability.** *The figure plots the percent of processes that run unaffected under disk failure from one busy hour from the HP trace. The degree of namespace replication is set aggressively to 32. Each line varies the amount of replication for "popular" directories; 1-way implies 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.*

availability. Specifically, we run a portion of the HP trace through a simulator with some number of failed disks, and record which percent of processes observed no I/O failure during the run. In Figure 3, we show 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. As the figure shows, 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 be available in order to run correctly. With popular directory replication, excellent availability under failure is possible. Fortunately, almost all of the popular files are found in "read only" directories; thus, wide-scale replication will not cause a write performance problem.

5 Semantic Knowledge

With a basic understanding of D-GRAID, 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 [37]. Understanding how the file system above utilizes the disk enables a D-GRAID to implement both graceful degradation under failure as well as quick recovery. The exact details of acquiring semantic knowledge within a disk or RAID system have been described elsewhere [37]; 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.

Unfortunately, static knowledge of file system layout and structures does not imply that the storage system has a complete understanding of file system behavior. Consider simple classification of blocks by their type. This process is straightforward for blocks that can be directly classified by location on disk. For example, in the Berkeley Fast File System (FFS) [22], the regions of disk that store inodes are fixed at file system creation; thus, any traffic to those regions is known to contain inodes, which can then be further interpreted. 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. This process of indirect classification is one example of the complexities involved in the implementation of semantically-smart disks.

In this paper, we extend understanding of semantically-smart disks by formally demonstrating both the extents and limits of semantic knowledge, and then later (in Section 6) show how a successful implementation can be achieved in spite of these limitations. Previous work required the file system to be mounted synchronously, for implementing complex functionality within the disk; we relax that requirement in this paper. Below, we first demonstrate that a semantic disk can never know for certain the type of a block when that block changes its type

during operation. We then demonstrate how a semantic disk can safely estimate block liveness.

5.1 File System Behaviors

To illustrate the extents and limitations of semantical knowledge within disks, we must first make assumptions about the behavior of file systems; how file systems reflect operations to disk has a strong impact on what types of knowledge can be garnered within the disk system. We assume the following generic behaviors of a file system. We believe that many if not all modern file systems adhere to these behavioral guidelines. We refer to a file system that exhibits all of these properties as a *typical* file system.

Dynamic Type: A Dynamic Type file system is willing to locate different types of blocks at the same physical location on disk over the lifetime of the file system; the prototypical example is a data block in a UNIX file system, which can be a user-data block, directory-data block, or indirect-pointer block at any given time. We assume that the type of the block is determined via information within another block, *e.g.*, an inode. We call the block that contains this information the *type-determining parent* of this block. A group of dynamically-typed blocks may share the same type-determining parent.

Arbitrary Order: An Arbitrary Order file system orders updates to the file system arbitrarily; hence, no particular update order can be relied upon to garner extra information about the nature of disk traffic. For example, in FFS, meta-data updates are forced to disk synchronously, and thus will arrive at the disk before the corresponding data. Other file systems are very careful in how they order updates to disk [9], and therefore some ordering could be assumed; however, to remain as general as possible, we avoid any such assumptions.

Delayed Update: A file system that delays updates to disk (often for performance reasons) is said to be a Delayed Update file system. Delays are found in writing data to disk in many file systems, including LFS [32], which buffers data in memory before flushing it to disk; this improves performance both by batching small updates into a single large one, and also by avoiding the need to write data that is created but then deleted. A contrast to delayed updates is a file system that immediately reflects file system changes to disk; for example, ext2 can be mounted synchronously to behave in this manner.

Hidden Operation: A file system that does not reflect all operations to disk is said to be a Hidden Operations file system. This property goes hand-in-hand with Delayed Updates, *e.g.*, a file system delays the disk update associated with file creation; a subsequent delete obviates the need to reflect the create to disk.

5.2 Dynamic Block Types

We now demonstrate that semantically-smart disks cannot be certain of the type of dynamically-typed blocks.

Assertion: *The type of a block which is dynamically typed cannot be positively identified through observation of the disk traffic underneath of a typical file system.*

Reasoning: *To determine the type of such a block, we must observe traffic until both the block and its type-determining parent have been seen. Because a typical file system arbitrarily orders blocks, the block and what seems to be its type-determining parent can be written to the disk in one of two orders.*

Case 1: Dynamically-typed first

1: Create file F_1, with type-determining parent P_1 and dynamically-typed block D of type T_1, denoted D_{T_1}. P_1 and D_{T_1} are in memory and dirty.
2: Write D_{T_1} to disk.
3: Delete F_1, freeing in-memory blocks P_1 and D_{T_1}.
4: Create file F_2, with type-determining parent P_2 and (reused) block D now with type T_2. Both P_2 and D_{T_2} are in memory and dirty.
5: Write P_2 to disk.
6: Wrongly conclude from P_2 that D_{T_1} is of type T_2.

Case 2: Type-determining parent first

1: As above.
2: Write P_1 to disk.
3: As above.
4: As above.
5: Write D_{T_2} to disk.
6: Wrongly conclude from P_1 that D_{T_2} is of type T_1.

Note that waiting after step 6 in case 1 (or 2) to see the updated version of D_{T_2} (or P_2) just leads us to case 2 (or 1), continuing the uncertainty.

Thus, no matter which order the disk observes the two blocks, the disk cannot know for sure whether a dynamically-typed block is of a particular type. □

This limitation has strong implications for D-GRAID, which must understand the contents of indirect-pointer blocks to implement graceful degradation.

5.3 Block Liveness

We also show that block liveness information can be safely utilized within a semantically-smart disk system. Specifically, we show that a block cannot be live on disk when the disk considers it dead, which is required to implement live-block recovery correctly. In general, blocks become "live" when a file is created or extended, and "dead" when a file is deleted or truncated.

For simplicity, we also assume the presence of a set of *allocation blocks*, which can be used to determine which blocks are free and which are allocated. For example, in Linux ext2, this is the set of inode and data bitmap blocks; in FAT, it is the file allocation table. Thus, for each block in the file system, there exists a unique allocation block that determines its liveness.

A semantic disk could use *conservative liveness estimation (CLE)* to classify live blocks. With CLE, a block is marked live when the disk sees a write to the block or a write to its allocation block with the block marked live.

Assertion: *With CLE, a live block on disk will not be classified as dead through observation of the disk traffic underneath of a typical file system.*

Reasoning: *We must consider two cases: block allocation and deletion. Assume block B is allocated or freed; its allocation block is A_B. We now consider allocation.*

Case 1a: Allocation

1: Allocate B and update A_B to indicate allocation. Both blocks are in memory and dirty.

2: Write A_B to disk.

3: Write B to disk.

Case 1b: Allocation

1: As above.

2: Write B to disk.

3: Write A_B to disk.

In case 1a, the allocation block reaches disk first, and thus B is considered live even before it reaches the disk. In case 1b, since the disk observes a write to a block B, CLE would conclude that the block has been allocated, even though the on-disk state of A_B indicates otherwise.

We now consider deletion.

Case 2: Deletion

1: Free B. Update A_B to indicate deletion. A_B is in memory and dirty.

2: Write A_B to disk.

When the write occurs, the disk is safe to consider block B free. If the block was previously allocated, the disk will consider it as such until the file system writes A_B to disk.

Because the disk can conservatively estimate liveness, a live on-disk block will not be classified as dead. □

We note that it is possible for the disk to consider a block live far longer than it actually is. This situation would arise if for example the file system is willing to write deleted blocks to disk, or if an allocation block can be dirtied but then cleaned without being forced to disk.

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 [32] 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 of the Linux ext2 file system. At the end of the section, we discuss differences in our implementation underneath of FAT.

6.1 Graceful Degradation

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

6.1.1 The Indirection Map

Just as any other SCSI-based RAID system, Alexander presents host systems with a volume to which they can issue block read and block write requests. 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 [7, 39, 40]. Unlike most of these other storage systems, this imap only maps every *live* logical file system block to all physical locations of a block (*i.e.*, all of its replicas). All other blocks are considered free (unmapped) and are candidates for use by the 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 [40]; currently Alexander uses a randomized selection.

6.1.3 Writes

In contrast to reads, write requests are much more complex to handle. Exactly how Alexander handles the write request depends on the *type* of the block that is written. We now discuss the different 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 inode and bitmap block types underneath of 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. For every newly added inode, D-GRAID selects a home site to layout blocks belonging to the inode. This selection of home site is done

to balance space allocation across physical disks. Currently, D-GRAID uses a greedy approach where it selects the home site that currently has the least amount of disk space committed.

However, 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.

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 logical-to-physical association in the imap, which maps 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 location(s). Thus, writes are deferred only for those blocks that are written *before* the corresponding owner inode blocks. If the inode arrives at the disk first, subsequent data writes will be already mapped and sent to disk immediately.

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; to understand which bits are new, the D-GRAID compares the newly written block with its old copy, a process referred to as block differencing. 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 [32]. Removing such blocks from the deferred list is important because in the case of freed blocks, Alexander may never observe an owning inode, thereby resulting in the block remaining in the deferred list indefinitely. 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 shown in Section 5. Thus, Alexander must handle incorrect type classification for data blocks (*i.e.*, file data, directory, and indirect blocks).

For example, consider indirect pointers. The D-GRAID must understand the contents of such blocks, because it uses the pointers therein to ensure placement of blocks of a file onto the correct home site. The main difficulty that arises due to our lack of perfect knowledge is that 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, the 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 create 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 has a window of vulnerability, because one of its blocks now lies 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. [2]

Thus, without any optimizations, D-GRAID will eventually move data into 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 [3], 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 a 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, and requires a great deal of bookkeeping. 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 generally 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 sequential access patterns.

To implement access-driven diffusion, Alexander divides up 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 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 [40], except that AutoRAID cannot identify data that is considered "dead" by the file system once written; in contrast, a D-GRAID system can use semantic knowledge to identify which blocks are free.

6.2 Live-block Recovery

To implement live-block recovery, the 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 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. 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.

The actual process of implementing live-block recovery is straightforward. Alexander simply locates the conservative bitmap table, and then uses that to build a list of blocks which need to be restored. Alexander proceeds through the list and performs the necessary data copies in order to restore all live data.

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

[2] 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 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 or stored in 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. Though 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 will 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. Assume a user data block becomes unavailable. When a process tries to read the block, 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 of ext2 and FAT-32; we initially thought the designs would diverge substantially. For example, FAT 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 FAT 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 FAT, there are no indirect (or doubly indirect, or triply indirect...) pointers to worry about. We also

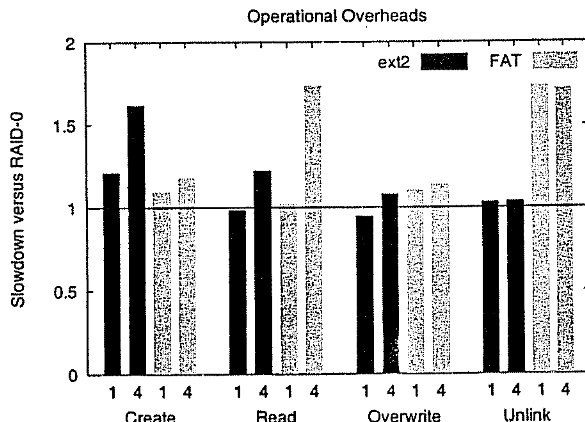

Figure 4: **Time Overheads.** *The figure plots the time overheads observed on D-GRAID Level 0 versus RAID Level 0 across a series of microbenchmarks. The tests are run on 1 and 4 disk systems. In each experiment, 3000 operations were enacted (e.g., 3000 file creations), with each operation on a 64 KB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

ran across the occasional odd behavior in the Linux implementation of FAT. 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 semantically-smart 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 include some baseline measurements of the FAT 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?*

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 500 MHz Pentium III and four 10K-RPM IBM disks.

Does Alexander work correctly? Alexander is a good deal more complex than simple RAID systems. To ensure

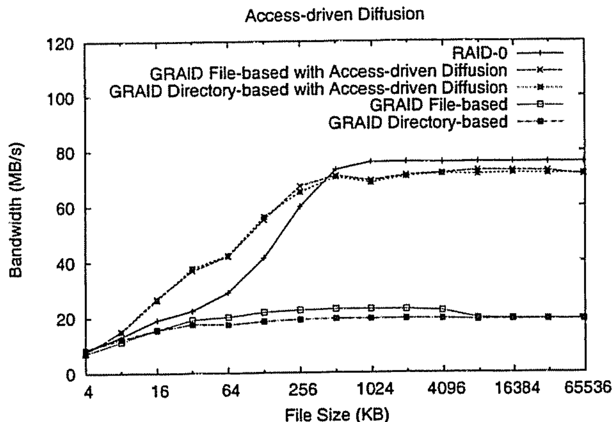

Figure 5: **Access-driven Diffusion.** *The figure presents the performance of D-GRAID Level 0 and standard RAID-0 under a sequential workload. In each experiment, a number of files of size x are read sequentially, with the total volume of data fixed at 64 MB. The y-axis shows the bandwidth achieved during the test, with and without access-driven diffusion. D-GRAID performs better for smaller files due to better physical block layout.*

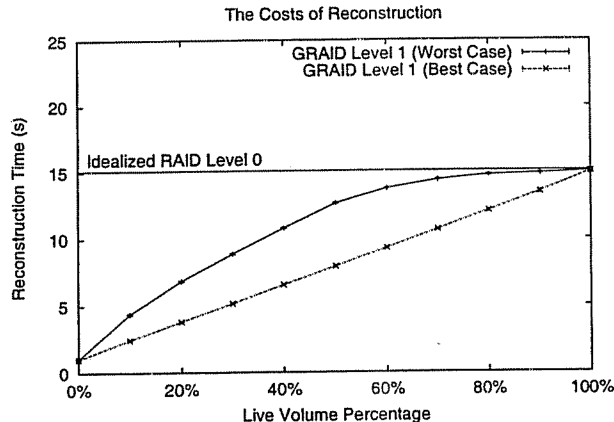

Figure 6: **Live-block Recovery.** *The figure shows the time to recover a failed disk onto a hot spare in a D-GRAID Level 1 (mirrored) system using live-block recovery. Two lines for D-GRAID are plotted: in the worst case, live data is spread across the entire 300 MB volume, whereas in the best case it is compacted into the smallest contiguous space possible. Also plotted is the recovery time of an idealized RAID Level 0.*

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 (eventually) detect which blocks were indirect blocks and move files and directories into their proper fault-isolated locations.

What time overheads are introduced? We first 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 4 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 heavily untuned and the infrastructure suffers from CPU contention with the host, we believe that these are worst case estimates of the overheads.

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 upto 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 5 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 to the diffused copies across all of the disks in the system.

How fast is live-block recovery? We now explore the potential improvement seen with live-block recovery. Figure 6 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 the 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 [34] could be applied to aid in recovery time, by moving all live data to a localized portion.

8 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 [29]. The Coda mobile file system also takes explicit care with regard to the directory tree [21]. 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.

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 [35]. 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 [19].

Finally, p2p systems such as PAST that place an entire file on a single machine have similar load balancing issues [33]. 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 [40] 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 [36], we learned that there could be a flexible trade-off between performance and reliability, and the value of delayed updates; we believe that there are many remaining components of D-GRAID where these approaches would help.

Much of RAID research has focused on different redundancy schemes. While early work stressed the ability to tolerate single-disk failures [2, 26, 27], later research introduced the notion of tolerating multiple-disk failures within an array [4]. We stress that our work is complementary to this line of research; traditional techniques can be used in order to ensure full file system availability up to a certain number of failures, and D-GRAID techniques ensure graceful degradation under additional failures. Further, as the semantic RAID taxonomy demonstrates, complex redundancy techniques could be applied differentially to data and meta-data, tailoring each scheme to the demands of the data type. A number of earlier works also emphasize the importance of hot sparing to speed recovery time in RAID arrays [16, 23, 26]. Our work on faster semantic recovery is also complementary to those approaches.

Finally, it should be noted that term "graceful degradation" is sometimes used to refer to the performance characteristics of redundant disk systems under failure [17, 30]. This type of graceful degradation is much different than what we discuss within this paper; indeed, none of those systems continues to operate when an unexpected number of failures occurs.

9 Summary and Conclusions

> "A robust system is one that continues to operate (nearly) correctly in the presence of some class of errors" *Robert Hagmann [15]*

A 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 found that a D-GRAID can be built underneath a standard block-based interface, 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 in 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 very careful in its assumptions about file system behavior, and hope that our model can serve as a guide to 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,

it is not surprising that file systems do not behave particularly well when part of a volume address space becomes unavailable. Perhaps because of its heritage as an OS for inexpensive hardware, we found that Linux file systems handled unexpected conditions fairly well. However, the exact model for how to deal with failure was inconsistent: data blocks could be missing and then reappear, but the same was 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 [31], 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.

We envision a number of interesting avenues for further research, including more intelligent semantic grouping, better policies and mechanisms for replication, and file system modifications that ease the construction of D-GRAID and other semantically-smart storage systems.

References

[1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *OSDI '02*, Boston, MA, December 2002.

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

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

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

[5] 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.

[6] 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.

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

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

[9] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, Monterey, CA, November 1994.

[10] 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.

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

[12] J. Gray, B. Horst, and M. Walker. Parity Striping of Disc Arrays: Low-cost Reliable Storage with Acceptable Throughput. In *VLDB 16*, pages 148–159, Brisbane, Australia, August 1990.

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

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

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

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

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

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

[19] 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.

[20] 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.

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

[22] 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.

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

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

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

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

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

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

[29] 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.

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

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

[32] 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.

[33] 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.

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

[35] 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.

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

[37] 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.

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

[39] 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.

[40] 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.

[41] 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.