Computer Sciences Department

Semantically-Smart Disk Systems

Muthian Sivathanu
Vijayan Prabhakaran
Florentina Popovici
Timothy Denehy
Andrea Arpaci-Dusseau
Remzi Arpaci-Dusseau

Technical Report ~~#1444~~ 1445

August 2002

UNIVERSITY OF WISCONSIN MADISON

Semantically-Smart Disk Systems

Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Computer Sciences Department
University of Wisconsin, Madison

Abstract: *We propose and evaluate the concept of a semantically-smart disk system (SDS). As opposed to a traditional "smart" disk, an SDS has detailed knowledge of how the file system above is using the disk, including information about the file system on-disk data structures and policies. An SDS exploits this knowledge to transparently improve performance or enhance functionality beneath a standard block read/write interface. To automatically acquire this knowledge, we introduce a tool (EOF) that can discover file-system specifics for certain types of file systems, and then show how an SDS can exploit this knowledge on-line to understand file-system behavior. We quantify the space and time overheads that are common in SDS's, showing that they are not excessive. We then study the issues surrounding SDS's by designing and implementing a number of prototype SDS's as case studies; each case study exploits knowledge of some aspect of the file system to implement powerful functionality beneath the standard SCSI interface. Overall, we find that a surprising amount of functionality can be embedded within an SDS, hinting at a future where disk manufacturers can compete on enhanced functionality and not simply cost-per-byte and performance.*

1 Introduction

> "To know that we know what we know, and that we do not know what we do not know, that is true knowledge." *Confucius*

As microprocessors and memory chips become smaller, faster, and cheaper, embedding processing and memory in peripheral devices becomes an increasingly attractive proposition [1, 15, 27, 33]. Placing processing power and memory capacity within a "smart" disk system allows functionality to be migrated from the file system into the disk (or RAID), thus providing a number of potential advantages over a traditional system. For example, when computation takes place near the data, one can improve performance by reducing traffic between the host processor and disk [1]. Further, such a disk system has and can exploit low-level information not typically available at the file-system level, including exact head position and block-mapping information [21, 29]. Finally, unmodified file systems can leverage these optimizations, enabling deployment across a broad range of systems.

Unfortunately, while smart disk systems have great promise, realizing their full potential has proven difficult. One causative reason for this shortfall is the *narrow interface* between file systems and disks [12]; the disk subsystem receives a series of block read and write requests that have no inherent meaning, and the data structures of the file system (*e.g.*, bitmaps for tracking free space, inodes, data blocks, directories, indirect blocks) are not exposed. Thus, research efforts have been limited to applying disk-system intelligence in a manner that is oblivious to the nature and meaning of file system traffic, *e.g.*, improving write performance by writing blocks to the closest free space on disk [11, 33].

To fulfill their potential and retain their utility, smart disk systems must become "smarter" while the interface to storage remains the same. Such a system must acquire knowledge of how the file system is using it, and exploit that understanding in order to enhance functionality or increase performance. For example, if the storage system understands which blocks constitute a particular file, it can perform intelligent prefetching on a per-file basis; if a storage system knows which blocks are currently unused by the file system, it can utilize that space for additional copies of blocks, for improved performance or reliability. We name a storage system that has detailed knowledge of file system structures and policies, a *Semantically-Smart Disk System (SDS)*, since it understands the meaning of the operations enacted upon it.

An important problem that must be solved by an SDS is that of "information discovery" – how does the disk learn about the details of file system on-disk data structures? The most straight-forward approach is to assume the disk has exact "white-box" knowledge of the file system structures (*e.g.*, with access to all relevant header files). However, in some cases such information will be unavailable or cumbersome to maintain. Thus, in this paper, we explore a "gray-box" approach [4], attempting where possible to automatically obtain such file-system specific knowledge within the storage system.

We develop and present a fingerprinting tool, EOF, that automatically discovers file-system layout through probes and observations. We show that by using EOF, a smart disk system can automatically discover the layout of a certain class of file systems, namely those that are similar to the Berkeley Fast File System (FFS) [22].

We then show how to exploit layout information to infer higher-level file-system behavior. The processes of *classification, association,* and *operation inferencing* include the ability to categorize each disk block (*e.g.*, data, inode, or bitmaps), to detect the precise type of each data block (*i.e.*, file, directory, or indirect pointers), to associate each data block with its inode, and to identify higher-level operations such as creating and deleting files and allocating and freeing blocks. An SDS can use some or all of these techniques to implement its desired functionality.

To prototype a smart disk system, we use a software infrastructure in which an in-kernel driver interposes on read and write requests between the file system and the disk. In our prototype environment, we can explore most of the challenges of adding functionality within an SDS, while adhering to existing interfaces and running underneath a stock file system. In this paper, we focus on the Linux ext2 file system, but also report on preliminary experience running underneath NetBSD FFS.

To understand the performance characteristics of an SDS, we study the overheads involved with fingerprinting, classification, association, and operation inferencing. Through microbenchmarks, we quantify costs in terms of both space and time, demonstrating that common overheads are not excessive.

Finally, to illustrate the potential of SDS's, we have implemented a number of case studies within our SDS framework: file-aware caching within the disk as an effective second-level cache [36], meta-data storage within a non-volatile RAM disk cache for performance and reliability [23, 32], aligning files with track boundaries to increase the performance of small-file operations [29], a secure-deleting disk system to ensure non-recoverability of data [16], and journaling within the storage system to speed crash recovery [17]. Through these case studies, we demonstrate that a broad range of functionality can be implemented within a semantically-smart disk system. In some cases, we also discuss how an SDS can tolerate imperfect information about file-system behavior, which is a key to building robust semantically-smart disk systems.

The rest of this paper is organized as follows. In Section 2, we discuss related work. We then discuss file-system fingerprinting in Section 3, on-line classification and association in Section 4, and on-line inferencing in Section 5. We evaluate our system in Section 6, and summarize our experience with the case studies in Section 7. Finally, we conclude in Section 8.

2 Related Work

2.1 Smart Disks and RAIDs

The related work on smart disks can be grouped into three categories. The first group assumes that the interface between the file and storage systems is fixed and cannot be changed, the category under which semantically-smart disk systems belong. Research in the second group proposes changes to the storage interface, requiring that file systems be modified to leverage this new interface. Finally, the third group proposes changes not only to the interface, but to the programming model for applications.

Fixed interfaces: The focus of this paper is on the integration of smart disks into a traditional file system environment. In this environment, the file system has a narrow, SCSI-like interface to storage, and uses the disk as a persistent store for its data structures. Loge [11] pioneered the concept of placing more processing capacity in the disk controller, improving performance by writing blocks near the current disk-head position. Wang *et al.*'s log-based programmable disk [33] extended this approach in a number of ways, namely quick crash-recovery and free-space compaction. Neither of these systems assume any knowledge of which blocks contain what type of file system data.

When storage system interfaces are more developed than that provided in the local setting, there are more opportunities for new functionality. The use of a network packet filter within the Slice virtual file service [3] allows Slice to interpose on NFS traffic in clients, and thus implement a range of optimizations (*e.g.*, preferential treatment of small files). Interposing on an NFS traffic stream is simpler than doing so on a local-disk block stream because the contents and fields of NFS packets are well-defined.

High-end RAID products are the perfect place for semantic smartness, since a typical enterprise storage system has substantial processing capabilities and memory (possibly persistent). For example, an EMC Symmetrix server contains up to eighty 333 MHz Motorola microprocessors and can be configured with up to 64 GB of memory [10]. Some high-end RAID systems currently leverage their resources to perform a bare minimum of semantically-smart behavior; for example, storage systems from EMC can recognize an Oracle data block and provide an extra checksum to assure that a block write (comprised of multiple disk sector writes) reaches disk atomically [14]. In this paper, we explore the acquisition and exploitation of more detailed knowledge of file system behavior within a smart disk system.

More expressive interfaces: Given that one of the primary factors that limits the addition of new functionality in a smart disk is the narrow interface between file

systems and storage, it is not surprising that there has been research that investigates changing this interface. We briefly highlight these projects. Mime investigates an enhanced interface in the context of an intelligent RAID controller [6]; specifically, Mime adds primitives to allow clients to control both when updates to storage become visible to other traffic streams and the commit order of operations. Logical disks expand the interface by allowing the file system to express grouping preferences with lists [8]; thus, file systems are simplified since they do not need to maintain this information. Finally, Ganger suggests that a reevaluation of this interface is needed [12], and outlines two case studies that span the boundary: track-aligned extents [29] (which we explore within this paper), and freeblock scheduling [21].

More radical "active" environments: In contrast to integration underneath a traditional file system, other work has focused on incorporating active storage into more radical environments. Recent work on "active disks" includes that by Acharya *et al.* [1], Riedel *et al.* [27], and Amiri *et al.* [2]. Much of the recent work focuses on new programming models for I/O-intensive parallel applications, and thus does not address the issue of retrofitting into existing systems.

2.2 Reverse Engineering

Reverse engineering techniques have been applied across a broad range of computer systems. For example, in programming language research, Hsieh *et al.* investigate the automatic extraction of bit-level instruction encodings by feeding permutations of instructions into an assembler and analyzing the resulting machine code [18]. A similar effort by Collberg automatically generates an efficient back-end for a compiler [7]. Our fingerprinting tool, discussed in Section 3, bears some similarity to these systems.

A specific type of reverse engineering known as *fingerprinting* has been successfully applied in identifying the policies of different systems. For example, fingerprinting has shown to be effective for identifying parameters within the TCP protocol [25], the low-level characteristics of disks [28], and characteristics of a real-time scheduler [26].

3 Inferring On-Disk Structures: Fingerprinting the File System

For a semantically smart disk to implement interesting functionality, it must be able to interpret the types of blocks that are being read from and written to disk and specific characteristics of those blocks. For SDS's to be practical, this information must be obtained in a robust manner that does not require substantial human involvement.

One approach for obtaining this knowledge is to place the onus on the developer of the semantically-smart disk system. In this "white box" approach, the SDS developer must have direct access to the source code for the file system and must directly hard-code the essential information into the SDS. The obvious drawback of this approach is that the SDS firmware must be updated whenever the file system changes. However, this approach may not be unreasonable in that production file system layouts do not change often (if at all). For example, the basic structure of FFS-based file systems has not changed since its introduction in 1984, a period of almost twenty years [22]; the Linux ext2 file system, introduced in roughly 1994, has had the exact same layout for its lifetime; finally, the ext3 journaling file system [31] is backwards compatible with ext2 layout and the new extensions to the FreeBSD file system [9] are backwards compatible as well. File system developers have strong motivation to keep on-disk structures the same over time, so that legacy file systems continue to operate without administrator or user intervention.

Although we believe this white-box approach can be legitimate for a semantically-smart storage system, we believe it is advantageous to have "gray-box" techniques that automatically infer the file system data structures. The advantages of this approach are many: no specific knowledge about the target file system is required when the SDS is developed; the assumptions made by the SDS about the target file system can be checked when it is deployed; little additional work is required to configure the SDS when it is installed; the SDS can be deployed in new environments with little or no difficulty. In this section, we explore how an SDS can automatically acquire layout information with fingerprinting software.

Automatically inferring layout information for an arbitrary file system is a challenging problem. Therefore, in this section, we describe an important first step: a utility, called EOF ("Extraction Of Filesystems"), that can extract layout information for FFS-based file systems. In this paper, we focus exclusively on Linux ext2. However by making only high-level assumptions within EOF, we believe that porting it to handle other FFS-based file systems will not be overly difficult; our preliminary experience with NetBSD FFS confirms this belief. In the future, we plan to explore automatic identification within a broader class of file systems.

3.1 Assumptions

In targeting file systems that are based on FFS, EOFmakes the following assumptions about the layout of file system data structures on the disk:

General: Disk blocks are statically and exclusively assigned to one of five categories: *data*, *inodes*, *bitmaps for free/allocated data blocks*, *bitmaps for free/allocated inodes*, and *summary information* (*e.g.*, superblock and group descriptors). EOF identifies the block addresses on disk allocated to each category.

Data blocks: A data block may dynamically contain either file data, directory listings, or pointers to other data blocks (*e.g.*, may be an indirect block). Data blocks are not shared across files.

Inode blocks: An inode block may contain multiple inodes; if N inodes fit in one inode block, then each inode consumes $\frac{1}{N}$-th of the space and no inode spans more than one block.

Inode structure: EOF identifies the presence (or absence) of the following fields within an inode: *size*, *blocks* (the number of data blocks allocated to this inode), *ctime* (the time at which the inode was last changed), *mtime* (the time at which the corresponding data was last changed), *dtime* (the deletion time), *links* (the number of links to this inode), *data pointers* (any number and combination of direct pointers and single, double, and triple indirect pointers) and *dir bits* (bits that change between file and directory inodes) All of the fields that we identify, with the exception of *dir bits* are assumed to be 32 bits. Indirect blocks are assumed to contain 32-bit pointers. EOF assumes that the definition of each inode field is static over time.

Bitmap blocks: Bits in the (data /inode) bitmap blocks have a one-to-one linear mapping to the data blocks/ inodes. The last bitmap block may not be entirely valid and is allocated last by the file system.

Directory data: Each record in a directory data block contains an entry name, the length of the record, the length of the file name, and the inode number of this file; each record may have other fields, including the type of the entry name (*e.g.*, file or directory).

3.2 Algorithm

The EOF software is used as follows. When a new file system is made on a SDS partition, EOF is run on the partition so that the SDS understands the context in which it is being deployed. The basic structure of EOF is that a user-level *probe process* performs operations on the file system, generating controlled traffic streams to disk. The SDS knows each of the high-level operations performed and the disk traffic that should result. By observing which file blocks are written and which bytes within blocks change, the SDS infers which blocks contain each type of file system data structures and which offsets within each block contain each type of field. The SDS can then either use this knowledge to configure itself or to assert that certain data structures are used.

The EOF algorithm has two distinct tasks: classifying blocks and identifying fields within the inode. Each task is divided into a number of phases: in phase 0, EOF isolates the summary blocks; in phase 1, EOF identifies data blocks and data bitmaps; in phase 2, EOF looks for inodes and inode bitmaps; in phase 3, EOF isolates most inode fields; in phase 4, EOF identifies the *dir bits* field in the inode; in phase 5, EOF identifies fields within a directory entry. The steps for classifying blocks are shown in Table 1, and the steps for identifying inode fields are shown in Table 2. Phases 0 and 5 are not shown due to space constraints.

After each step, the probe process performs a `sync` to flush data structures to disk so that the SDS can associate disk blocks with particular operations; however, `sync` is used sparingly since it increases the running time of EOF. An out-of-band channel between the EOF and the SDS (e.g. an ioctl) is used to communicate step and phase boundaries. Each table shows the block traffic or inode fields that the SDS expects to observe written or changed on each operation. To classify data blocks, the SDS looks for a known pattern that the probe process writes to all files. To classify all other blocks and all inode fields, EOF employs the general technique of *isolation*, in which the SDS looks for a unique, unclassified block that was written within a step, across a set of steps, or in some steps but not others. For example, the only type of block expected to be written in both step 2 and step 3 is the file inode; therefore, the block that appeared in both steps must be a file inode. The table captions describe more details of EOF.

3.3 Assertion of Assumptions

The EOF algorithm implicitly relies on a set of assumptions at each step. To be robust to file systems that do not meet these assumptions, EOF has mechanisms to detect when any of the assumptions fail, and to identify the file system as non-supported, instead of conveying erroneous information to the SDS. For example, if more than two blocks are written during step 3 of phase 1 in Table 1, EOF would detect it as a violation of assumptions. We tested EOF on a range of file systems like msdos, vfat, reiserfs and ext3 and EOF identified violations in each case.

4 Exploiting Structural Knowledge: Classification and Association

The key advantage of an SDS is its ability to identify and utilize important properties of each block on the disk. These properties can be determined through direct and indirect classification as well as through association. With *direct classification*, blocks are easily identified by their

Phase	Step	Probe operation	Traffic: Dir Data	Dir Inode	Data Bitmap	Inode Bitmap	Indirect Data	File Data	File Inode	New Inferences
1	1	create small file	X	X/P[1]	X	X		X/i	X	File data: Content (Known pattern)
	2	rename file	X	X/P					X	None
	3	rewrite file contents	i					X/P	X/i	File inode: Isolate (3) Dir data: Elimination in (2)
	4	append large data			X		X	X/i	X/P	Many data blocks: Content
	-	repeat 1-4 until all data blocks filled								Most data blocks on disk: Content[2]
	4'	repeat of append			X/I		X/I	X/I	X/P	Bitmaps: Blocks written by several files Indirect data blocks: Blocks written by only one file[3]
	5	cleanup (delete files)								None
2	1	create empty files	X	X	X/P	X			X	None
	2	chmod files		X/I					X/I	Inodes: Isolate (2)
	3	rename files	X/i	X/P		I			X/P	Dir data: Isolate (3) Inode Bitmap: Isolate (1)
	4	cleanup (delete files)								None

Table 1: **Steps performed by EOF to identify static structures on disk.** *The* Traffic *column indicates the block traffic generated and inferred by this operation: an* X *indicates that a block of that type is written; an* i *shows that a block of this type is identified in this step; an* I *is used when all blocks of that type are identified; a* P *is used when the block was previously identified. Traffic to summary blocks is ignored for simplicity. The last column explains how the SDS component of EOF makes the inferences; when isolation is used to identify a block, the numbers in parenthesis indicate the steps of the traffic involved. Notes are as follows. 1) The directory inode block of the root inode is obtained in phase 0 by observing traffic during a* `mount` *2) Not quite all data blocks are identified; additional data blocks allocated to the parent directory remain unclassified. 3) Care is taken to use small enough files such that no single file can fill a bitmap block; the last bitmap block is a more involved special case since a smaller than expected file can completely fill it.*

Phase	Step	Probe operation	Traffic: size	blocks	ctime	mtime	dtime	links	pointers	dir bits	New Inferences
3	1	create small file[1]	X	X	X	X	X	X	X	X	None
	2	overwrite data			X	X					None
	3	append byte	X/I		X	X					Size: Isolate (3, not 2)
	4	create link			X/I	I		X/I			Ctime: Isolate (3,4) Links: Isolate (4) Mtime: Isolate (3, not 4)
	5	repeatedly seek + append[2]	X/P	X/I	X/P	X/P			X/I		Blocks: Changes each time Pointers: Change for one append Indirection Level: Size of next append for new pointer
	6	delete both links	X/P	X/P	X/P		X/I	X/P			Dtime: Isolate (6)
4	1	create many files	X/P	X/P	X/P	X/P	X/P	X/P	X/P	X	None: Track identical bits in all files
	2	create many dirs	X/P	X/P	X/P	X/P	X/P	X/P	X/P	X	None: Track identical bits in all dirs
	-	repeat 1,2 if necessary	X/P	X/P	X/P	X/P	X/P	X/P	X/P	I	Dir bits: Look for different bits across files and dirs

Table 2: **Steps performed by EOF to identify fields within inodes.** *The* Traffic *column indicates fields in the inode that are changed from the previous time the inode was written. Notes are as follows. 1) Small files are created until there is one that does not share an inode block with the parent directory. 2) The seek amount is grown progressively rather than writing every block in order to improve performance. The seek distance starts at one block; each time a new indirection level is detected, the distance is increased by the size handled by that indirection level. The detection of the version number field in the inode is not shown due to space constraints.*

location on disk. With *indirect classification*, blocks are identified only with additional information; for example, to identify directory data or indirect blocks, the corresponding inode must also be examined. Finally, with *association*, a data block and its inode are connected.

In many cases, an SDS also requires functionality to identify when a change has occurred within a block. This functionality is implemented via block differencing. For example, to infer that a data block has been allocated, a single-bit change in the data bitmap must be observed. Change detection is potentially one of the most costly operations within an SDS for two reasons. First, to compare the current block with the last version of the block, the SDS may need to fetch the old version of the block from disk; however, to avoid this overhead, a cache of blocks can be employed. Second, the comparison itself may be expensive: to find the location of a difference, both blocks must be scanned and each byte compared with the corresponding byte in the other block. We quantify these costs in Section 6.

4.1 Direct Classification

Direct classification is the simplest and most efficient form of on-line block identification for an SDS. The SDS determines the type of the block by performing a simple bounds check to calculate into which set of block ranges a particular block falls; of course, these ranges can be obtained with either white-box or fingerprint information. For example, in an FFS-based file system, the superblock, bitmaps, inodes, and data blocks are identified using this technique.

4.2 Indirect Classification

Indirect classification is required when the type of a block can vary dynamically and thus simple direct classification cannot precisely determine the type of block. For example, in FFS-based file systems, indirect classification is used to determine whether a data block is file data, directory data, or some form of indirect pointers (*e.g.*, a single, double, or triple indirect block). To illustrate these concepts we focus on how directory data is differentiated from file data; the steps for identifying indirect blocks versus pure data are similar.

4.2.1 Identifying Directory Data

The basic challenge in identifying whether a data block belongs to a file or a directory is to track down the inode that points to this data and check whether its type is a file or a directory. To perform this tracking, the SDS observes all inode traffic to and from the disk: when a directory inode is observed, the corresponding data block numbers are inserted into a hash table. The SDS removes data blocks from the hash table by observing when those blocks are freed (*e.g.*, by using block differencing on the bitmaps). Thus, when the SDS must later identify a data block as a file or directory, in general, its presence in this table indicates that it is directory data. We now discuss two complications.

First, the SDS cannot always guarantee that it can correctly identify blocks as files or directories. Specifically, when a data block is not present in the hash table, the SDS infers that the data corresponds to a file; however, in some cases, the directory inode may not have yet been seen by the SDS and as a result is not yet in the hash table. Such a situation may occur when a new directory is created or when new blocks are allocated to existing directories; if the file system does not guarantee that inode blocks are written before data blocks, the SDS may incorrectly classify newly written data blocks. This problem does not occur when classifying data blocks that are read. In this case, the file system must read the corresponding inode block before the data block (to find the data block number); thus, the SDS will see the inode first and correctly identify subsequent data blocks.

Whether or not transient misclassification is a problem depends upon the functionality provided in the SDS. For instance, if an SDS simply caches directory blocks for performance, it can tolerate temporary inaccuracy. However, if the SDS requires accurate information for correctness, there are two ways it can be ensured. The first option is to guarantee that the file system above writes inode blocks before data blocks; this is true by default in the original FFS and is true in Linux ext2 when mounted in synchronous mode. The second option is to buffer writes until the time when the classification can be made; this *deferred classification* occurs when the corresponding inode is written to disk or when the the data block is freed, as can be inferred by monitoring data bitmap traffic.

Second, the SDS may perform excess work if it obliviously inserts all data blocks into the hash table whenever a directory inode is read and written since this inode may have recently passed through the SDS, already causing the hash table to be updated. Therefore, to optimize performance, the SDS can infer whether or not a block has been added (or modified or deleted) since the last time this directory inode was observed, and thus ensure that only those blocks are added to (or deleted from) the hash table. This process of *operation inferencing* is described in detail in Section 5.

4.2.2 Identifying Indirect Blocks

The process for identifying indirect blocks is almost identical to that for tracking directory data blocks. In this case, the SDS tracks new indirect block pointers in all inodes

being read and written. The SDS maintains another hash table of all single, double, and triple indirect blocks numbers and uses it to quickly identify if a data block is an indirect block.

4.3 Association

The most useful association is to connect data blocks with their inodes; for example, this allows the size or creation date of a file to be known by the SDS. Association can be achieved with a simple but space-consuming approach. Similar to indirect classification, the SDS observes all inode traffic and inserts the data pointers into an address-to-inode hash mapping table. One concern about such a table is size; for accurate association, the table grows in proportion to the number of unique data blocks that have been read or written to the storage system since the system booted. However, if approximate information is tolerated by the SDS (as we explore in Section 7), the size of this table can be bounded.

5 Detecting High-Level Behavior: Operation Inferencing

Block classification and association provide the SDS with an efficient way for identifying special kinds of blocks; however, operation inferencing is necessary to understand the semantic meaning of the changes observed in those blocks. In this section, we outline how the SDS can identify logical file system operations by looking at the changes made to those blocks.

One challenge with operation inferencing is that the SDS must distinguish between blocks which have a valid "old version" and those that do not. For instance, when a newly allocated directory block is written, it should not be compared to the old contents of the block since the block contained arbitrary data. To identify when to use the old versions, the SDS uses a simple insight: when a meta-data block is written without being read, the old contents of the block are not relevant. To detect this situation, the SDS maintains a hash table of meta-data block addresses that have been read sometime in the past. Whenever a meta-data block is read, it is added to this list; whenever the block is freed (as indicated by a block bitmap reset), it is removed from the list. For example, when a block allocated to a data file is freed and reallocated to a directory, the block address will not be present in the hash table, and therefore the SDS will not use the old contents to detect changes.

For illustrative purposes, in this section we examine how the SDS can infer file create and delete operations. Our current approach is tailored to identifying operations in the Linux ext2 file system; we plan to explore operation inferencing across a broader range of file systems in the near future.

5.1 File Creates and Deletes

There are two steps in identifying file creates and deletes. The first step is the actual detection of a create or delete; the second step is determining the inode that has been affected. We briefly describe three different detection mechanisms and the corresponding logic for determining the associated inode.

The first detection mechanism involves the inode block itself. Whenever an inode block is written, the SDS examines it to determine if an inode has been created or deleted. A valid inode has a non-zero modification time and a zero deletion time. Therefore, whenever the modification time changes from zero to non-zero or the deletion time changes from non-zero to zero, it means the corresponding inode was newly made valid, *i.e.*, created. Similarly, a reverse change indicates a newly freed inode, *i.e.*, a deleted file. A second indication is a change in the version number of a valid inode, which indicates that a delete followed by a create occurred. In both cases, the inode number is calculated using the physical position of the inode on disk; note that the on-disk inode structure does not contain the inode number.

The second detection mechanism involves the inode bitmap block. Whenever a new bit is set in the inode bitmap, it indicates that a new file has been created corresponding to the inode number represented by the bit position. Similarly, a newly reset bit indicates a deleted file.

The update of a directory block is a third indication of a newly created or deleted file. When a directory data block is written, the SDS examines the block for changes from the previous version. If a new directory entry (`dentry`) has been added, the name and inode number of the new file can be obtained from the `dentry`; in the case of a removed `dentry`, the old contents of the `dentry` contain the name and inode number of the deleted file.

Given that any of these three changes indicate a newly created or deleted file, the choice of the appropriate mechanism (or combinations thereof) depends on the functionality being implemented in the SDS. For example, if the SDS must identify the deletion of a file, immediately followed by the creation of another file with the same inode number, the inode bitmap mechanism does not help, since the SDS may not observe a change in the bitmap if the two operations are grouped due to a delayed write in the file system. In such a case, using modification times and version numbers is more appropriate. Similarly, if the name of the newly created or deleted file must be known, the directory block-based solution is the most efficient.

5.2 Other File System Operations

The general technique of inferring logical operations by observing changes to blocks from their old versions can help detect other file system operations as well. We note that in some cases, for a conclusive inference on a specific logical operation, the SDS must observe correlated changes in multiple meta-data blocks. For example, the semantically-smart disk system can infer that a file has been renamed when it observes a change to a directory block entry such that the name changes but the inode number stays the same; note that the version number within the inode must stay the same as well. Similarly, to distinguish between the creation of a hard link and a normal file, both the directory entry and the inode of the file must be examined.

6 Evaluation

In this section, we answer three important questions about our SDS framework. First, what is the cost of fingerprinting the file system? Second, what are the time overheads associated with classification, association, and operation inferencing? Third, what are the space overheads? Before proceeding with the evaluation, we first describe our experimental environment.

6.1 Platform

To prototype an SDS, we employ a software-based infrastructure. Our implementation inserts a pseudo-device driver into the kernel, which is able to interpose on traffic between the file system and the disk. Similar to a software RAID, our prototype appears to file systems above as a pseudo-device upon which a file system can be mounted.

The primary advantage of our prototype is that it observes the same information and traffic stream as an actual SDS, with no changes to the file system above. However, our current infrastructure differs in two important ways from a true SDS. First, and most importantly, our prototype does not have direct access to low-level drive internals; using such information is thus made more difficult. Second, the performance characteristics of the microprocessor (a Pentium-III) and memory system may be different than an actual SDS; however, high-end storage arrays already have significant processing power, and over time this processing capacity will likely trickle down into less-expensive storage systems.

We have experimented with our prototype in both the Linux 2.2 and NetBSD 1.5 operating systems, underneath of the ext2 and FFS file systems, respectively; however, our focus is on the Linux platform. All experiments in this paper are performed on a 550 MHz Pentium-III processor with either an IBM 9LZX or Quantum Atlas 10K III disk.

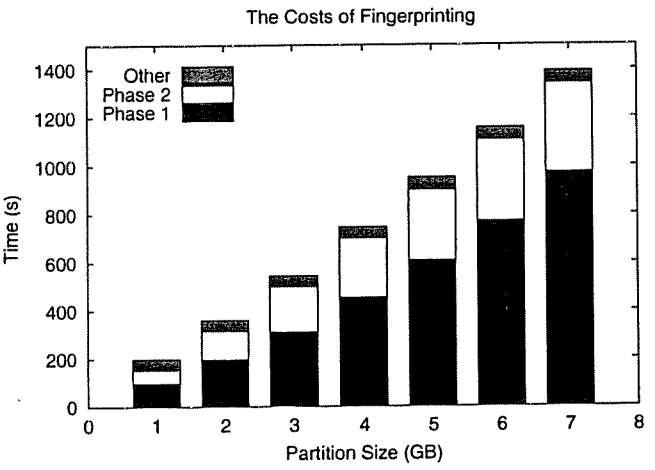

Figure 1: **The Costs of Fingerprinting.** *The figure presents the time breakdown of the fingerprinting on the Linux ext2 file system on an IBM 9LZX disk. Along the x-axis, we vary the size of the partition that is fingerprinted, and the y-axis shows the time taken per phase.*

6.2 Off-line: Layout Discovery

In this subsection, we show that the time to run the fingerprinting tool, EOF, is reasonable for modern disks. Given that EOF only needs to run once for each new file system, the runtime of EOF does not determine the common case performance of an SDS; however, we do not want the runtime of EOF to be prohibitive, especially as disks become larger. One potential solution is parallelism: we believe that the EOF has many parallelizable components, which could be exploited to reduce run-time on disk arrays.

Figure 1 presents a graph of the time to run EOF on a single-disk partition as the size of the partition is increased. The graph shows that Phase 1, which determines the locations of data blocks and data bitmaps, and Phase 2, which determines the locations of inode blocks and inode bitmaps, dominate the total cost of fingerprinting. The time for these two phases increases roughly linearly with the size of the partition, requiring approximately 150 seconds per gigabyte. The other phases of EOF, which determine the fields within an inode and directory entry, require the same time regardless of the partition size.

6.3 On-line: Time Overheads

Classification, association, and operation inferencing are potentially costly operations for an SDS. In this subsection, we employ a series of microbenchmarks to illustrate the various costs of these actions. The results are presented in Table 3. For each action and microbenchmark we consider two cases. In the first case, the file system is mounted synchronously, ensuring that meta-data operations reach the SDS in order and thus allowing the SDS to guarantee correct classification with no additional effort. In the second case, the file system is mounted asyn-

	Indirect Classification		Block-Inode Association		Operation Inferencing	
	Syn	Asyn	Syn	Asyn	Syn	Asyn
$Create_0$	1.7	3.2	1.9	3.3	33.9	3.2
$Create_{32}$	60.6	3.8	324.4	16.4	279.7	3.8
$Delete_0$	4.3	3.6	6.7	3.9	50.9	3.6
$Delete_{32}$	37.8	6.9	80.1	28.8	91.0	6.9
Mkdir	56.3	8.6	63.6	11.1	231.9	8.6
Rmdir	49.9	106.2	57.8	108.5	289.4	106.2

Table 3: **SDS Time Overheads.** *The table breaks down the costs of indirect classification, block-inode association, and operation inferencing. Different microbenchmarks are applied (one per row), to stress various aspects of each action. The Create benchmark creates 1000 files, of size 0 or 32 KB, and the Delete benchmark similarly deletes 1000 such files. The Mkdir and Rmdir benchmarks create or remove 1000 directories, respectively. Each result in the table presents the average cost per operation in microseconds. Across all experiments, the IBM 9LZX was used, with the Linux ext2 file system mounted in either synchronous (***Syn***) or asynchronous (***Asyn***) mode.*

	Indirect Classification	Block-Inode Association	Operation Inferencing
$NetNews_{50}$	5,880	104,265	6,258
$NetNews_{100}$	7,203	139,419	7,875
$NetNews_{150}$	7,959	166,761	8,988
$PostMark_{20}$	294	38,619	1,071
$PostMark_{30}$	294	56,385	1,386
$PostMark_{40}$	294	79,905	1,701
Andrew	30	302	114

Table 4: **SDS Space Overheads.** *The table presents the space overheads of the structures used in performing classification, association, and operation inferencing, under three different workloads (NetNews, PostMark, and the modified Andrew benchmark). Two of the workloads (NetNews and PostMark) were run with different amounts of input, which correspond roughly to the number of "transactions" each generates (i.e., $NetNews_{50}$ implies 50,000 transactions were run). Each number in the table represents the maximum number of hash-table entries recorded during the benchmark run. In this experiment, we used the Linux ext2 file system (asynchronous mode), and the IBM 9LZX.*

chronously; in this case, to guarantee correct classification and association the SDS must perform operation inferencing.

From our experiments, we make a number of observations. First, most operations tend to cost on the order of tens of microseconds. Although some of the operations do require more than 100 μs, most of this cost is due to a a per-block cost; for example, operation inferencing in synchronous mode with the $Create_{32}$ workload takes roughly 280 μs, which corresponds to a 34 μs base cost (as seen in the $Create_0$ workload) plus a cost of approximately 30 μs for each 4 KB block. Thus, although the costs rise as file size increases, the SDS incurs only a small overhead compared to the actual disk write. Second, in most cases, performance when the ext2 file system is run in asynchronous mode is much higher than when run in synchronous mode. In asynchronous mode, numerous updates to meta-data are batched and thus the costs of block differencing are amortized; in synchronous mode, each meta-data operation is reflected through to the disk system, incurring much higher overhead in the SDS. Third, we observe that in synchronous mode, classification is less expensive than association which is less expensive than inferencing; an SDS should take care to employ only those that are needed to implement the required functionality.

6.4 On-line: Space Overheads

SDS's require additional memory to perform classification, association, and operation inferencing; in particular, hash tables are required to track mappings between data blocks and inodes whereas caches are needed to help with block differencing. In this subsection we quantify the memory overheads induced in an SDS for a variety of workloads.

Table 4 presents the number of entries in each hash table to support classification, association, and operation inferencing; each hash table entry requires approximately eight bytes. The sizes are the maximum reached during the run of a particular workload: NetNews [30], PostMark [20], and the modified Andrew benchmark [24]. For NetNews and PostMark, we vary workload size, as described in the caption.

From the table, we see that the dominant memory overhead occurs in an SDS performing block-inode association. Whereas classification and operation inferencing require table sizes that are proportional to the number of unique meta-data blocks that pass through the SDS, association requires information on every unique data block that passes through. In the worst case, an entry is required for every data block on the disk, corresponding to 1 MB of memory for every 1 GB of disk space. Although the space costs of tracking association information are high, we believe they are not prohibitive.

Block differencing requires that a cache of "old" blocks is maintained. Thus, the performance of the system is sensitive to the size of this cache; if the cache is too small, each difference calculation must first fetch the old version of the data from disk. To avoid the extra I/O, the size of the cache must be roughly proportional to the active meta-data working set. For example, for the $PostMark_{20}$ workload, we found that the SDS cache should contain approximately 650 4 KB blocks to hold the working set.

7 Case Studies

In this section, we describe five case studies, each implementing new functionality in an SDS. These SDS's are ordered roughly in increasing complexity, from a relatively straightforward caching SDS to an intricate journaling SDS. Because of space limitations, we cannot fully describe each of the case studies in this paper; instead, we

	Read Small Files	Re-Read Small Files
FFS	18.3 MB/s	18.3 MB/s
+Caching SDS	18.3 MB/s	85.8 MB/s

Table 5: **File-Aware Caching.** *The table shows the time it took to read and then re-read 64 64-KB files; between the read and re-read, a scan clears the OS buffer cache. We compare NetBSD FFS on a standard IBM 9LZX disk or on an SDS.*

highlight the functionality each case study implements, present a brief performance evaluation, and then analyze the complexity of implementing said functionality within an SDS-based system. One theme we explore within this section is the usage of "approximate" information: when can an SDS afford to be wrong about its view of the file system?

7.1 The Case Studies

File-Aware Caching:

Simple LRU management of the disk cache is likely to cache only data already present in the file system cache [34, 36], and thereby waste memory in the storage system. This waste is particularly onerous in storage arrays, due to their large amounts of memory. To illustrate the ease with which an SDS can implement a complementary replacement policy, we target a common workload in which accesses to small files are interleaved with an occasional scan of a large file; it is well-known that this workload performs poorly with LRU caches since the large file completely removes the many small files from the cache. The Caching SDS exploits knowledge of file size to selectively cache only "small" files and their inodes, where a small file is defined to be one that only uses direct block pointers; given that we have implemented this case study on NetBSD, this corresponds to files that are less than 96 KB. Of course, other more sophisticated policies targeted for different workloads, such as the Multi-Queue second-level cache management policy [36], may instead be implemented within the SDS.

To implement file-aware caching, the SDS identifies which blocks belong to small files using indirect classification; in this case, the classification hash table holds blocks corresponding to small files. As described previously, when the file system is mounted asynchronously, this may cause the SDS to misclassify blocks in those rare cases when the file inode is written to disk after the data blocks. However, in this case study, it is acceptable for the SDS to occasionally not recognize blocks belonging to small files. Table 5 shows that for this workload, the Caching SDS substantially improves the performance of multiple reads to small files, since the small files stay in the disk cache.

Non-volatile Memory: In this case study, we demonstrate how an SDS can exploit battery-backed memory to provide improved performance and reliability. An

	LFS Benchmark			PostMark
	Create	Read	Delete	
FFS	10.0	0.04	11.0	230.0
+NVRAM SDS	0.14	0.04	0.1	19.0

Table 6: **NVRAM.** *The left three columns of the table show the time in seconds to complete each phase of the LFS microbenchmark (the LFS benchmark creates, reads and deletes 1000 1K files). The right column shows the total time in seconds for the PostMark benchmark, run with 5000 files, 5000 transactions, and 71 directories. This experiment took place on the IBM 9LZX running NetBSD.*

NVRAM SDS can use knowledge of the file system to store meta-data and some file data in memory, improving the access time of meta-data intensive workloads while preserving file system reliability. In contrast to the Conquest [32] and Hermes file systems [23], which both stipulate large changes to the file system to exploit non-volatile memory, no changes to the file system are needed in the semantically-smart disk system approach.

The NVRAM SDS keeps all meta-data (bitmaps, inodes, indirect blocks, and directories) in NVRAM. Inodes and bitmaps are identified by their location on the disk. Pointer blocks and directory data blocks are identified with indirect classification, which can occasionally miss blocks. Here again we exploit the fact that approximate information is adequate; the SDS writes unclassified blocks to disk and not NVRAM, until it observes the corresponding inode. To track meta-data blocks, the NVRAM SDS uses a map to record their in-core location. The NVRAM SDS also periodically flushes meta-data from NVRAM to disk, increasing overall reliability since disks are likely to be more reliable than battery-backed RAM [13]. Table 6 shows that an unmodified NetBSD file system can achieve performance orders of magnitude greater, by using the NVRAM SDS.

Track-Aligned Extents: As proposed by Schindler *et al.* [29], track-aligned extents (traxtents) can improve disk access times by placing medium-sized files within tracks and thus avoiding track-switch costs. Given the detailed level of knowledge that a traxtents-enabled file system requires of the underlying disk (*i.e.*, the mapping of logical block numbers to physical tracks), traxtents are a natural candidate for implementation within an SDS, where this information is readily obtained.

The fundamental challenge of implementing traxtents in an SDS instead of the file system is in adapting the *policies* of the file system outside of the file system; specifically, a Traxtent SDS must influence file system allocation and prefetching, *e.g.*, mid-sized files must be allocated such that consecutive data blocks do not span tracks boundary and accesses to aligned files must be in track-sized units.

There are three components of interest within the Traxtent SDS implementation. First, when the bitmap blocks are first read by the file system, the SDS marks the bitmap

	No SDS Prefetching	+ SDS Prefetching
ext2	10.3 MB/s	10.2 MB/s
+Traxtent SDS	12.2 MB/s	14.2 MB/s

Table 7: **Traxtents.** *The table shows the bandwidth obtained when reading 100 random files of size 328 KB. We examine default and track-aligned allocation combined with and without prefetching at the SDS. This experiment took place on the Atlas 10K III running Linux 2.2, with ext2 mounted asynchronously.*

corresponding to the last block in each track as allocated, (a similar technique is used by Schindler *et al.*). Although this wastes a small portion of the disk, this "fake" allocation influences the file system to allocate files such that they do not span tracks. Second, if the file system still decides to allocate a file across tracks, the SDS dynamically remaps those blocks to a track-aligned locale, similar to the block remapping of Loge and other smart disks [11, 33]. One major difference is that the SDS only remaps blocks that are a part of mid-sized files that benefit from track-alignment, whereas non-semantically aware disks cannot make such a distinction. Third, the Traxtent SDS performs additional prefetching to ensure accesses are not smaller than a track. Linux ext2 prefetches very few blocks when a file is initially read; therefore, when the traxtent SDS observes a read to the first block of a track-aligned file, it requests the remainder of the track and places the data blocks in its cache.

The Traxtent SDS relies upon one piece of exact information for correctness: the location of bitmap blocks, which it marks to "trick" the file system into track-aligned allocation. However, given that this information is static, it can be obtained reliably with EOF and with little performance cost at runtime. The indirect classification of file data as belonging to medium-sized files can be occasionally incorrect, since their remapping is only for performance and not correctness. Table 7 shows that the Traxtent SDS results in a modest improvement in bandwidth for medium-sized files, especially when these files are prefetched by the SDS.

Secure Deletion: With advanced magnetic force scanning tunneling microscopy (STM), a person with physical access to a drive (and a lot of time) can potentially extract sensitive data that the user had "deleted" [16]. In this case study, we explore a "secure-deleting" SDS, that is, a disk that guarantees that file data from deleted files is truly unrecoverable. Previous approaches have placed such functionality within the file system by over-writing deleted file blocks multiple times with various patterns [5]. However, this does not guarantee that the data is removed from the disk; as pointed out by Gutman [16], other copies of various data blocks may exist, due to bad-block remapping or storage system performance optimizations [35]. An SDS is the only locale where a secure delete can be implemented, since it can ensure no stray copies of data exist.

	Delete	PostMark
ext2	24.0	103.0
+Secure-deleting SDS_2	46.9	128.0
+Secure-deleting SDS_4	56.9	142.0
+Secure-deleting SDS_6	63.6	192.0

Table 8: **Secure Deletion.** *The table shows the time in seconds to complete a Delete microbenchmark and the PostMark benchmark on the Secure-deleting SDS. The Delete benchmark deletes 1000 files, whereas the PostMark benchmark performs 1000 transactions. Each row with the Secure-deleting SDS shows performance with a different number of over-writes (2, 4, or 6). This experiment took place on the IBM 9LZX with Linux 2.2, with ext2 mounted synchronously.*

Because of the nature of this case study, approximate or incorrect information about which blocks have been deleted is not acceptable. The Secure-deleting SDS recognizes deleted blocks through operation inferencing and then overwrites those blocks with different data patterns a specified number of times. Since the file system may reallocate these blocks to a different file and possibly write the block with fresh contents in the meantime, the SDS tracks deleted blocks and queues writes to those blocks until the overwrite has finished.

Table 8 shows the overhead incurred by an SDS, as a function of the number of over-writes; the more over-writes performed, the less likely the data will be recoverable. Although a noticeable price is paid for the secure-delete functionality, this loss may be acceptable to highly-sensitive applications requiring such security. Performance could be further improved by delaying the secure-overwrite until the disk is idle, instead of performing it immediately; freeblock scheduling may further minimize the performance impact [21].

Journaling: The final case study is the most complex – the SDS implements journaling underneath of an unsuspecting file system. We view the Journaling SDS as an extreme case which helps us to understand the amount of information we can obtain at the disk level. Unlike most of the other case studies, almost all of the information about the file system required by the Journaling SDS is exact.

Due to space limitations, we only present a brief summary of the implementation. The fundamental difficulty in implementing journaling in an SDS arises from the fact that at the disk, transaction boundaries are blurred. For instance, when a file system does a file create, the file system knows that the inode block, the parent directory block, and the inode bitmap block are updated as part of the single logical create operation, and hence these block writes can be grouped into a single transaction in a straight-forward fashion. However, the SDS sees only a stream of meta-data writes, potentially containing interleaved logical file system operations. The challenge lies in identifying dependencies among those blocks and handling updates as atomic transactions.

	LFS Benchmark		
	Create	Read	Delete
ext2 (2.2/sync)	63.9	0.32	20.8
ext2 (2.2/async)	0.28	0.32	0.03
ext2 (2.4/async)	0.25	0.13	0.05
ext3 (2.4)	0.47	0.13	0.26
ext2 (2.2/sync)+SDS	0.95	0.33	0.24

Table 9: **Journaling.** *The table shows the time to complete each phase of the LFS microbenchmark in seconds with 1000 32-KB files. All data is the average of 30 runs; deviations were low (less than 5%). This experiment took place on the IBM 9LZX running Linux.*

EOF Fingerprinting			
Probe process	1463		
In-SDS component	2453		
SDS Infrastructure		**Case Studies**	
Initialization	395	File-aware cache	45
Hashtable/cache	260	NVRAM	95
Direct classification	195	Traxtents	1320
Indirect classification	75	Secure delete	80
Association	15	Journaling	2440
Operation inferencing	1105		

Table 10: **Code Complexity of SDS's.** *The number of lines of code for various aspects of the SDS are presented.*

As a result, the Journaling SDS maintains transactions at a coarser granularity than what a journaling file system might. The basic approach is to buffer meta-data writes in memory and write them to disk only when the in-memory state of the meta-data blocks constitute a consistent meta-data state. This is logically equivalent to performing incremental in-memory fsck's on the current set of dirty meta-data blocks and writing them to disk when the check succeeds. When the current set of dirty meta-data blocks form a consistent state, they are treated as a single atomic transaction, thereby ensuring that the on-disk meta-data contents either remain at the previous (consistent) state or are fully updated with the next consistent state. One benefit of these more coarse-grained transactions is that by batching commits, performance may be improved over more traditional journaling systems.

To guarantee bounded loss of data after crash, the Journaling SDS limits the time that can elapse between two successive journal transaction commits. A journaling daemon wakes up periodically after a configurable interval and takes a copy-on-write snapshot of the dirty blocks in the cache and the dependency information. After this point, subsequent meta-data operations update a new copy of the cache, and therefore cannot introduce additional dependencies in the current epoch.

For correct operation, the current Journaling SDS implementation assumes the file system has been mounted synchronously. To be robust, the SDS requires a way to verify that this assumption holds and to turn off journaling otherwise. Since the meta-data state written to disk by the Journaling SDS is consistent regardless of a synchronous or asynchronous mount, the only problem imposed by an asynchronous mount is that the SDS might miss some operations that were reversed (*e.g.*, a file create followed by a delete); this would lead to dependencies that are never resolved and indefinite delays in the journal transaction commit. To avoid this problem, the Journaling SDS looks for suspicious multiple changes in meta-data blocks when only a single change is expected (*e.g.*, multiple inode bitmap bits change as part of a single write) and turns off journaling in such cases. As a fall-back, the Journaling SDS also monitors the elapsed time since the last meta-data commit; if dependencies prolong the commit by more than a certain time threshold, it suspects an asynchronous mount and aborts. To check correctness, we ran `fsck` on a file system recovered by the Journaling SDS after a crash and verified that no inconsistencies are reported.

The performance of the Journaling SDS is summarized in Table 9. Although this SDS requires the file system to be mounted synchronously, its performance is similar to the asynchronous versions since the semantically-smart disk system delays writing meta-data to disk. In the read test the SDS has similar performance to the base file system (ext2 2.2), and in the delete test, it has similar performance to the journaling file system, ext3. It is only during file creation that the SDS pays a significant cost relative to ext3; the overhead of block differencing and hash table operations do have a noticeable impact. Since the purpose of this case study is to demonstrate the range of functionality that can be implemented in an SDS we feel the performance penalty is not severe.

7.2 Complexity Analysis

We briefly explore the complexity of implementing software for an SDS. Table 10 shows the number of lines of code for each of the components in our system and the case studies. From the table, one can see that most of the complexity is found in the EOF tool and the operation inferencing code. Most of the case studies are trivial to implement on top of this base infrastructure; however, Traxtent SDS and Journaling SDS require a few thousand lines of code. Thus, we conclude that including this type of functionality in an SDS is quite pragmatic.

8 Conclusions

> "Beware of false knowledge; it is more dangerous than ignorance." *George Bernard Shaw*

In a recent article on "Wise Drives", Dr. Gordon Hughes, Associate Director of the Center for Magnetic Recording Research, writes in favor of "smarter" drives, stressing their great potential for improving storage system performance and functionality [19]. However, he

believes a new interface between file systems and storage is required: "For widespread uses, its [a drive's] input/output and command requirements need to appear in the interface specification. In short, there must be an industry consensus that the task is of general interest and offers market opportunities for multiple computer and drive companies." Hughes' comments illustrate the difficulty of new interfaces – they require wide-scale industry agreement, which eventually limits creativity to only those inventions that fit into an existing interface framework.

With information about how the file system uses the disk and low-level knowledge of drive internals, an SDS sits in an ideal location to implement powerful pieces of functionality that neither a disk nor a file system can implement on its own, enabling innovation behind existing interfaces. In this paper, we have demonstrated that underneath of a particular class of FFS-like file systems, file-system information can be automatically gathered and then exploited to implement functionality in drives that heretofore had to be implemented in the file system or could not be implemented at all.

Many challenges remain, including understanding the generality and robustness of SDS-based techniques across a broader range of file systems. Can more sophisticated file systems, including journaling and log-based approaches, be probed to reveal their inner workings? Can approximate information be exploited more fully to implement interesting new functionality? We believe the answer is yes, but only through further research and experimentation will the final answer be elicited.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *ASPLOS VIII*, San Jose, California, October 1998.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *USENIX*, June 2000.

[3] D. Anderson, J. Chase, and A. Vahdat. Interposed Request Routing Scalable Network Storage. *TOCS*, 20(1), 2002.

[4] A. Arpaci-Dusseau and R. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *SOSP '01*, Oct. 2001.

[5] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *USENIX Security*, August 2001.

[6] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: A high performance parallel storage device. Technical Report HPL-CSP-92-9, HP Labs, 1992.

[7] C. S. Collberg. Reverse Interpretation + Mutation Analysis = Automatic Retargeting. In *PLDI '97*, June 1997.

[8] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The Logical Disk: A New Approach to Improving File Systems. In *SOSP '93*, Asheville, NC, December 1993.

[9] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *FREENIX*, June 2002.

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

[11] R. M. English and A. A. Stepanov. Loge: A Self-Organizing Disk Controller. In *USENIX*, Jan. 1992.

[12] G. R. Ganger. Blurring the Line Between Oses and Storage Devices. TR SCS CMU-CS-01-166, December 2001.

[13] G. A. Gibson and D. A. Patterson. Designing Disk Arrays for High Data Reliability. *JPDC*, 17(1/2), Jan./Feb. 1993.

[14] J. Gray. *Personal Communication*, 2002.

[15] J. Gray. Storage Bricks Have Arrived. In *Invited Talk: FAST '02*, Monterey, California, January 2002.

[16] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *USENIX Security*, July 1996.

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

[18] W. C. Hsieh, D. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *USENIX*, June 2001.

[19] G. F. Hughes. Wise Drives. *IEEE Spectrum*, August 2002.

[20] J. Katcher. PostMark: A New File System Benchmark. NetApp TR-3022, October 1997.

[21] C. Lumb, J. Schindler, and G. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *FAST '02*, January 2002.

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

[23] E. Miller, S. Brandt, and D. Long. HeRMES: High-Performance, Reliable MRAM-Enabled Storage. In *HotOS VIII*, Schloss Elmau, Germany, May 2001.

[24] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *USENIX*, June 1990.

[25] J. Padhye and S. Floyd. On Inferring TCP Behavior. In *SIGCOMM '01*, San Deigo, CA, August 2001.

[26] J. Regehr. Inferring Scheduling Behavior with Hourglass. In *FREENIX*, Monterey, CA, June 2002.

[27] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining. In *VLDB*, 1998.

[28] J. Schindler and G. R. Ganger. Automated Disk Drive Characterization. TR CMU-CS-99-176, 1999.

[29] J. Schindler, J. L. Griffin, C. R. Lumb, and G. R. Ganger. Track-aligned Extents: Matching Access Patterns to Disk Drive Characteristics. In *FAST '02*, January 2002.

[30] K. Swartz. The Brave Little Toaster Meets Usenet. In *LISA '96*, pages 161–170, Chicago, Illinois, October 1996.

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

[32] A. Wang, P. Reiher, G. Popek, and G. Kuenning. Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. In *USENIX*, June 2002.

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

[34] T. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *USENIX*, June 2002.

[35] X. Yu, B. Gum, Y. Chen, R. Wang, K. Li, A. Krishnamurthy, and T. Anderson. Trading Capacity for Performance in a Disk Array. In *OSDI '00*, 2000.

[36] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *USENIX*, June 2001.