Semantically-Smart Disk Systems

Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy,
Andrea C. Arpaci-Dusseau, and 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 system, including information about the on-disk data structures of the file system. 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 structure 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 an SDS, showing that they are not excessive. We then study the issues surrounding SDS construction by designing and implementing a number of prototypes 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 has become an increasingly attractive proposition [1, 19, 32, 40]. 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 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 [26, 35]. 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 [16]; 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, and 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 [15, 40].

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.*, 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) [27].

We then show how to exploit layout information to infer higher-level file-system behavior. The processes of *classification, association,* and *operation inferencing* refer to the ability to categorize each disk block (data, inode, bitmaps, or superblock) and detect the precise type of each data block (file, directory, or indirect pointer), to associate each data block with its inode or other relevant information, and to identify higher-level operations such as file creation and deletion. 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 and ext3 file systems, as well as 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 semantically-smart storage systems, we have implemented a number of case studies within our SDS framework: aligning files with track boundaries to increase the performance of small-file operations [35], using information about file-system structures to implement more effective second-level caching schemes with both volatile and non-volatile memory [43], a secure-deleting disk system that ensures non-recoverability of deleted files [20], and journaling within the storage system itself to improve crash recovery time [21]. 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 demonstrate how an SDS can tolerate imperfect information about the file system, 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, classification and association in Section 4, and operation inferencing in Section 5. We evaluate our system in Section 6, and present case studies in Section 7. We conclude in Section 8.

2 Related Work

The related work on smart disks can be grouped into three categories. The first group assumes that the interface between file and storage systems is fixed and cannot be changed, the category under which an SDS belongs. 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. An early example of a smart disk controller is Loge [15], which harnessed its processing capabilities to improve performance by writing blocks near the current disk-head position. Wang *et al.*'s log-based programmable disk [40] extended this approach in a number of ways, namely quick crash-recovery and free-space compaction. Neither of these systems assume or require any knowledge of file system structures.

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 SCSI-disk block stream because the contents of NFS packets are well-defined.

High-end RAID products are the perfect place for semantic smartness, because a typical enterprise storage system has substantial processing capabilities and memory capacity. 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 [14]. 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 sector writes) reaches disk atomically [7]. In this paper, we explore the acquisition and exploitation of more detailed knowledge of file system behavior.

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 [9]; 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 [11]; thus, file systems are simplified since they do not need to maintain this information. E×RAID exposes per-disk information to an informed file system (namely, I·LFS), providing performance optimizations, more control over redundancy, and improved manageability of storage [12]. Finally, Ganger suggests that a reevaluation of this interface is needed [16], and outlines two relevant case studies: track-aligned extents [35] (which we explore within this paper), and freeblock scheduling [26].

More recent work in the storage community suggests that the next evolution in storage will place disks on a more general-purpose network and not a standard SCSI bus [17]. Some have suggested that these network disks export a higher-level, object-like interface [18], thus moving the responsibilities of low-level storage management from the file system into the drives themselves. Although the specific challenges would likely be different in this context, the fixed object-based interface between file systems and storage will likely provide an interesting avenue for further research into the utility of semantic awareness.

New programming environments: In contrast to integration underneath a traditional file system, other work has focused on incorporating active storage into entirely new parallel programming environments. Recent work on "active disks" includes that by Acharya *et al.* [1], Riedel *et al.* [32], and Amiri *et al.* [2]. Much of this research focuses on how to partition applications across host and disk CPUs to minimize data transferred across system busses.

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 an SDS to be practical, this information must be obtained in a robust manner that does not require human involvement. We consider three alternatives for obtaining this information.

The first approach directly embeds knowledge of the file system within the SDS; thus, the onus of understanding the target file system is placed on the developer of the SDS. The obvious drawbacks are that SDS firmware must be updated whenever the file system is upgraded and the SDS is not robust to changes in the target file system.

With the second approach, the target system informs the SDS of its data structures at run-time; in this case, the responsibilities are placed on the target file system. There are numerous disadvantages with this approach as well. First, and most importantly, the target system must be changed; either the file system (or some other process with access to the same information) must directly communicate with the SDS. Second, a new communication channel outside of existing protocols must be added between the target system and the SDS. Finally, it may be difficult to ensure that the specification communicated to the SDS matches the actual file system implementation.

In the third approach, the SDS automatically infers the file system data structures. The benefits 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. We believe that this approach has the most potential for a semantically-smart storage system; thus, we explore how an SDS can automatically acquire layout information with fingerprinting software.

Automatically inferring file system structures bears similarity to several other research efforts in reverse-engineering. For example, researchers have shown that both bit-level machine instruction encodings [22] and the semantic meaning of assembly instructions [10] can be deduced. Others have also developed techniques to identify parameters of the TCP protocol [30], to extract low-level characteristics of disks [34, 38], to determine OS buffer-cache policies [8], and to understand the behavior of a real-time CPU scheduler [31].

3.1 Assumptions

Automatically inferring layout information for an arbitrary file system is a challenging problem. As an important first step, we have developed a utility, called EOF ("Extraction Of Filesystems"), that can extract layout information for FFS-like file systems, either with or without journaling capabilities. We have verified that EOF can identify the data structures employed by Linux ext2 and ext3 as well as NetBSD FFS. Furthermore, EOF should be able to understand a future FFS-like file system that adheres to the following assumptions about the layout of data structures on disk:

General: Disk blocks are statically and exclusively assigned to one of five categories: *data*, *inodes*, *bitmaps for free/allocated data blocks and/or inodes*, *summary information* (*e.g.*, superblock and group descriptors), and *log data*. 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 (*i.e.*, an indirect block). Data blocks are not shared across files. EOF identifies the structure of directory data

as well; EOF assumes that each record in a directory data block contains at least the length of the record, the entry name, the length of the entry name, and the inode number for the entry. Each field in a directory entry is assumed to be a multiple of 8 bits. EOF assumes that indirect blocks contain 32-bit pointers.

Inode blocks: An inode block contains N inodes where each inode consumes exactly $1/N$-th of the block. EOF assumes that the definition of each inode field is static over time. EOF identifies the location (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), *generation number*, *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). With the exception of *dir bits*, all of the fields that we identify are by default assumed to be a multiple of 32 bits; however, if multiple fields are identified within the same 32 bits (*e.g.*, the *blocks* and *links* fields), then the size of each field is assumed to be the largest multiple of 8 bits that does not lead to overlapping fields.

Bitmap blocks: Bitmaps for data and inodes may either share a single block or be placed in separate blocks. Bits in the (data /inode) bitmap blocks have a one-to-one linear mapping to the data blocks/inodes. The last bitmap block does not have to be entirely valid.

Log data: The log data used by a journaling file system is managed as a circular, contiguous buffer. We make no assumptions about the contents of the log, although we may look into doing so in the future.

The feasibility of inferring on-disk data structures depends upon the assumption that production file systems change slowly over time (if at all). This assumption is likely to hold, given that file system developers have strong motivation to keep on-disk structures the same, so that legacy file systems can continue to operate. Examining file systems of the past and present further corroborates this belief. For example, the on-disk structure of the FFS file system has not changed in nearly 20 years [27]; the Linux ext2 file system has had the same layout since conception; the ext3 journaling file system is backward compatible with ext2 [39]; extensions to FreeBSD FFS are designed so as to avoid on-disk changes [13].

3.2 Algorithm Overview

The EOF software is used as follows. When a new file system is made on an 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 use this knowledge to configure itself, simultaneously verifying that the target file system behaves as expected.

The SDS must be able to correlate the traffic it observes with the file system operations performed by the probe process. This correlation requires two pieces of functionality. First, the probe process must ensure that all blocks from an operation have been flushed out of the file system cache and written to the SDS. To ensure this, the probe process unmounts the file system; however, unmounting (and re-mounting) is used sparingly since it increases the running time of EOF. Second, the probe process must occasionally inform the SDS that a specific operation has ended. The probe process communicates to the SDS by writing a distinct pattern to a *fencepost* file; the SDS looks for this known pattern in the resulting traffic to find the message from the probe process.

Two general techniques are used within EOF to identify blocks and inode fields. First, to identify data blocks, the SDS always looks for a known pattern that the probe process writes in test files. Second, to classify all other blocks and fields, the SDS attempts to isolate a unique, unclassified block that is written by one operation, across a set of operations, or by some operations but not by others.

3.3 Algorithm Phases

EOF is composed of five phases. First, EOF isolates the summary blocks and the log file. Next, EOF identifies data blocks and data bitmaps. Then, EOF looks for inodes and inode bitmaps. After all blocks have been classified, EOF isolates the inode fields. Finally, EOF identifies the fields within directory entries.

3.3.1 Bootstrapping (Phase 0)

The goal of bootstrapping is to isolate the blocks that are frequently written in later phases so that they can be filtered from the blocks of interest. Thus, phase 0 isolates summary blocks, the log file, and inode and data blocks for the fencepost file, the test directory, and a few test files.

First, the probe process creates the fencepost file and a number of test files within a test directory; the SDS identifies the data blocks associated with each file by searching for the known patterns. Second, EOF identifies the blocks belonging to the log file, if it exists. In this step, the probe process synchronously appends data with a known pattern

to one of the test files. The SDS observes many blocks of meta-data; those blocks that are written to in a circular pattern belong to the log (if no block traffic matches this pattern, then EOF infers that the file system does not perform journaling). Third, EOF identifies the summary blocks; the probe process unmounts the file system and the written blocks that have not been classified as log data are identified as summary blocks.

To isolate the inode blocks that are repeatedly written, the probe process performs a `chmod` on the fencepost file, the test directory, and the test files; in each case, only the inode of each is written, allowing it to be classified. The data blocks belonging to the test directory are identified by changing the name of each test file; these blocks are the only previously unidentified blocks written. Finally, to determine if separate bitmap blocks are used for data and inode blocks (*i.e.*, as in Linux ext2 and ext3) or if a single bitmap block is shared between both (*i.e.*, as in NetBSD FFS), EOF creates a new file; whether the SDS observes one or two unclassified blocks allows it to determine whether bitmap blocks are shared or kept separate for data and inodes. To simplify our presentation, in the remainder of our discussion we consider only the case where data and inode bitmaps are in separate blocks; however, EOF correctly handles the shared case, in which case, EOF also isolates the specific bits in the shared bitmap block devoted to inode or data block state.

3.3.2 Data and Data-bitmap blocks (Phase 1)

EOF continues by identifying all the blocks on disk containing either data or data bitmaps. To isolate these blocks, the probe process appends a few blocks of data with a known pattern to each of the test files. All blocks that do not match the known pattern and are not yet classified are assumed to be either data-bitmap blocks or indirect-pointer blocks. EOF differentiates between the two by inferring that blocks written by two different files must be data-bitmap blocks. Care is taken to create small enough files such that no single file fills a bitmap block; the last bitmap block is a special case since a smaller than expected file can completely fill it. To cleanup from this phase, the test files are deleted.

3.3.3 Inodes and Inode-bitmap blocks (Phase 2)

Identifying the inodes and their bitmaps requires creating many new files. Two distinct steps are required. First, the probe process creates many new files, which causes both the inodes and inode bitmaps to be modified. Second, the probe process performs a `chmod` on the files, which causes the inodes but not the inode bitmaps to be written. Thus, the inodes and inode bitmaps can be distinguished from each other. This phase also calculates the size of each inode; this is performed by recording the number of times each block is identified as an inode and dividing the block size by the observed number of inodes in a block.

3.3.4 Inode Fields (Phase 3)

At this point, EOF has classified all blocks on disk as belonging to one of the five categories of data structures. The next phase identifies fields within inodes by observing those fields that do or do not change across operations. For brevity, we do not describe how EOF infers the blocks, links, and generation number fields.

The first inode fields that EOF identifies are the file size and times; this requires five steps. First, the probe process creates a file; the SDS stores the inode data to compare it to the inode data written in the next steps. Second, the probe process overwrites the file data; the only inode fields that change are those related to time. Third, the probe process appends a small amount of data to the file such that a new data pointer is not added; at this point, the size field can be identified as the only data that changed in step 3 but not step 2. Fourth, the probe process performs an operation to change the inode without changing the file data (*e.g.*, adding a link or changing the permissions); this allows the SDS to isolate *mtime* (which is not changed in this step) from *ctime* (which is changed). Finally, the file is deleted so that the deletion-time field is observed.

EOF next identifies the location and the level of the data pointers in the inode. The probe process repeatedly appends to a file while the SDS observes which bytes in the inode change (other than those that changed in the previous step). EOF infers the location of indirect pointers (and so forth) by observing when an additional "data" block is written and no additional pointer is updated in the inode. To improve performance, rather than write every block, the probe process seeks a progressively larger amount: the seek distance starts at one block and increases by the size handled by the currently detected indirection level.

Finally, EOF isolates the inode bit fields that designate directories. The probe process alternately creates files and directories. The SDS keeps two histograms: one for file and one for directory inodes; in the histogram, EOF records the count of times each bit in the inode type was zero. To determine the directory fields, EOF isolates all bits that were always 0 for files and always 1 for directories (and vice versa). These bits and their corresponding values are then considered to identify files versus directories. Soft link bits are identified in a similar manner.

3.3.5 Directory Entries (Phase 4)

In its final phase, EOF identifies the structure of entries within a directory. First, EOF infers the offsets of the entry name and the name length. To do this, the probe pro-

cess creates a file with a known name; the SDS searches for this name in the directory data block as well as the field designating the length of this name. For validation, this file is deleted and the step is repeated numerous times for filenames of different lengths. Second, EOF finds the location of the record length, using the assumption that the length of the last record contains the remaining space for the directory data block and that this length is reduced when a new record is added. Thus, the probe process creates additional files and the SDS simply records the offsets that change in the previous entries. Finally, the offset of the inode number is found using the assumption that each directory contains an entry for itself (*i.e.*, "."). In this step, the probe process creates two empty directories; the SDS isolates the inode offset by recording the differences across the data blocks of those two directories.

3.4 Assertion of Assumptions

The major challenge with automatic inferencing is to ensure that the SDS has correctly identified the behavior of the target file system. To be robust to a new file system not meeting these assumptions, EOF has mechanisms to detect when an assumption fails; in this case, the file system is identified as non-supported and the SDS operates correctly, but without using semantic knowledge. For example, if more blocks than expected are written in a specific step, or if specific blocks are not observed, EOF detects this as a violation. We have verified that violations are identified appropriately when EOF is run upon non-FFS file systems (*e.g.*, msdos, vfat, and reiserfs)."

An additional benefit of using EOF to configure an SDS is that file system bugs may be identified. For example, running EOF on ext3 in Linux 2.4 isolated two bugs. First, the SDS observed incomplete traffic in key steps; this problem was tracked back to an ext3 bug in which data written within 30 seconds prior to an unmount is not always flushed to disk [28]. Second, the probe process noted an error when all of the inodes were allocated; in this case, ext3 incorrectly marks the file system as dirty [25]. Thus, EOF enables checks of the file system that are not easily obtained with other methods.

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 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, each byte in the new block must be compared with the corresponding byte in the old 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. In an FFS-like 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-like file systems, indirect classification is used to determine whether a data block is file data, directory data, or some form of indirect pointer block (*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.

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 *snoops* on 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). When the SDS must later identify a block as a file or directory block, its presence in this table indicates that it is directory data. We now discuss two complications with this approach.

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 likely tolerate a 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 FFS (before soft updates [36]) and 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 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.

Identifying indirect blocks: The process for identifying indirect blocks is almost identical to that for identifying directory data blocks. In this case, the SDS tracks new indirect block pointers in all inodes being read and written. By maintaining a hash table of all single, double, and triple indirect block addresses, an SDS can determine 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 snoops on all inode traffic and inserts the data pointers into an address-to-inode hash 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, 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. We now outline how an SDS can identify file system operations by observing certain key changes.

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 hence the SDS will not use the old contents.

For illustrative purposes, in this section we examine how the SDS can infer file create and delete operations. The discussion below is specific to ext2, although similar techniques can be applied to other FFS-like file systems.

5.1 File Creates and Deletes

There are two steps in identifying file creates and deletes. The first is the actual detection of a create or delete; the second is determining the inode that has been affected. We 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 (on-disk inodes do not contain inode numbers).

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 file's inode 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 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 three 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, because the SDS runs on the same system as the host OS, there may be interference due to competition for resources; in our initial case studies, we do not believe this to be of prime importance. Third, the performance characteristics of the microprocessor and memory system may be different than an actual SDS; however, high-end storage arrays already have significant processing power, and this processing capacity will likely trickle down into lower-end storage systems.

We have experimented with our prototype SDS in the Linux 2.2, Linux 2.4, and NetBSD 1.5 operating systems, underneath of the ext2, ext3, and FFS file systems, respectively. Most experiments in this paper are performed on a processor that is "slow" by modern standards, a 550 MHz Pentium III processor, with either an 10K-RPM IBM 9LZX or 10K-RPM Quantum Atlas III disk. In some experiments, we employ a "fast" system, comprised of a 2.6 GHz Pentium IV and a 15K-RPM Seagate Cheetah disk, to gauge the effects of technology trends.

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

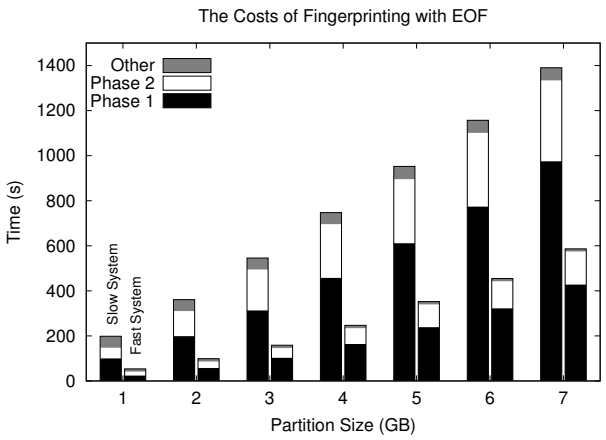

Figure 1: **The Costs of Fingerprinting.** *The figure presents the time breakdown of the fingerprinting on both the "slow" system (IBM disk), and the "fast" system, both running underneath Linux ext2. Along the x-axis, we vary the size of the partition that is fingerprinted, and the y-axis shows the time taken per phase.*

that the time-consuming components of EOF are parallelizable, which would 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. We show performance results for both the "slow" system with the IBM disk and the "fast" system. 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 linearly with the size of the partition, requiring approximately 190 seconds per GB on the slow system, and 81 seconds per GB on the fast system. Comparing performance across the two systems, we conclude that increases in sequential disk performance directly improve EOF fingerprinting time. The other phases require a small amount of time regardless of 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 of our experiments on an SDS underneath of Linux ext2 are presented in Table 1. 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; synchronous mounting in Linux ext2 is quite similar to traditional FFS in its handling of meta-data updates. In the second case, the file system is mounted asynchronously; in this case, to guarantee correct classification and association the SDS must perform operation inferencing. The microbenchmarks perform basic file system operations, including file and directory creates and deletes, and we report the per-file or per-directory overhead of the action that is under test.

From our experiments, we make a number of observations. First, most operations tend to cost on the order of tens of microseconds per file or directory. Although some of the operations do require nearly 300 μs to complete, most of this cost is due to 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 per-block overhead compared to the actual disk writes, each of which may take some number of milliseconds to complete. Second, in most cases, the overheads when the ext2 file system is run in asynchronous mode are much lower than when run in synchronous mode. In asynchronous mode, numerous updates to meta-data blocks 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 actions that are needed to implement the desired functionality.

6.4 On-line: Space Overheads

An SDS may require additional memory to perform classification, association, and operation inferencing; specifically, hash tables are required to track mappings between data blocks and inodes whereas caches are needed to implement efficient block differencing. We now quantify these memory overheads under a variety of workloads.

Table 2 presents the number of bytes used by each hash table to support classification, association, and operation inferencing. The sizes are the maximum reached during the run of a particular workload: NetNews [37], PostMark [24], and the modified Andrew benchmark [29]. 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

	Indirect Classification		Block-Inode Association		Operation Inferencing	
	Sync	**Async**	**Sync**	**Async**	**Sync**	**Async**
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 1: **SDS Time Overheads.** *The table breaks down the costs of indirect classification, block-inode association, and operation inferencing. Different microbenchmarks (one per row) 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 presents the average overhead per operation in μs (i.e., how much extra time the SDS takes to perform classification, association, or inferencing). The experiments were run upon the "slow" system with the IBM 9LZX disk, with Linux ext2 mounted synchronously (***Sync***) or asynchronously (***Async***).*

	Indirect Classification	Block-Inode Association	Operation Inferencing
NetNews_{50}	68.9 KB	1.19 MB	73.3 KB
NetNews_{100}	84.4 KB	1.59 MB	92.3 KB
NetNews_{150}	93.3 KB	1.91 MB	105.3 KB
PostMark_{20}	3.45 KB	452.6 KB	12.6 KB
PostMark_{30}	3.45 KB	660.7 KB	16.2 KB
PostMark_{40}	3.45 KB	936.4 KB	19.9 KB
Andrew	360 B	3.54 KB	1.34 KB

Table 2: **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 bytes stored in the requisite hash table during the benchmark run (each hash entry is 12 bytes in size). The experiment was run on the "slow" system with Linux ext2 in asynchronous mode on the IBM 9LZX disk.*

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. Further, if memory resources are scarce, the SDS can choose to either tolerate imperfect information (if possible), or swap portions of the table to disk.

In addition to the hash tables needed to perform classification, association, and operation inferencing, a cache of "old" data blocks is useful to perform block differencing effectively; recall that differencing is used to observe whether pointers have been allocated or freed from an inode or indirect block, to check whether time fields within an inode have changed, to detect bitwise changes in a bitmap, and to monitor directory data for file creations and deletions. 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 block 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. When the cache is smaller, block differencing operations often go to disk to retrieve the older copy of the block, increasing run-time for the benchmark by roughly 20%.

7 Case Studies

In this section, we describe our case studies, each implementing new functionality in an SDS that would not be possible to implement within a drive or RAID without semantic knowledge. Some of these case studies could be built into the file system proper; however, implementing "file-system like" functionality in the storage system is one the many advantages of semantic intelligence, as it allows storage-system manufacturers to augment their products with a much broader range of capabilities.

Due to space limitations, we cannot fully describe each of the case studies in this paper; instead, we highlight the functionality each case study implements, present a brief performance evaluation, and conclude by analyzing the complexity of implementing said functionality within an SDS. Each performance evaluation is included to demonstrate that interesting functionality can be implemented effectively within an SDS; we leave more detailed performance studies as future work. One theme we explore within this section is the usage of "approximate" information, *i.e.*, scenarios in which an SDS can be wrong in its understanding of the file system.

7.1 The Case Studies

Track-Aligned Extents: As proposed by Schindler *et al.* [35], 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 to 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

	Without Prefetching	With Prefetching
ext2	10.3 MB/s	10.2 MB/s
+Traxtent SDS	12.2 MB/s	14.2 MB/s

Table 3: **Track-Aligned Extents.** *The table shows the bandwidth obtained when reading 100 files in a randomized order. Each file is roughly the size of a track, in this case 328 KB. We examine both default and track-aligned allocation, varying whether track-sized prefetching is enabled within the SDS. This experiment was run upon the "slow" system running Linux 2.2 with the ext2 file system mounted asynchronously upon the Quantum Atlas III disk.*

	TPC-B_{20}	TPC-B_{100}
FFS	25.04	45.27
+LRU SDS	26.52	48.58
+File-Aware Caching SDS	3.88	20.58

Table 4: **File-Aware Caching.** *The table shows the time in seconds it takes to execute 20,000 and 100,000 TPC-B transactions. In all experiments, transactions first run to warm up the system; then a large scan is run, followed by another series of transactions, which are timed. The table compares NetBSD FFS on a standard disk, on an SDS with a 100 MB LRU-managed cache, and on an SDS with a 100 MB file-aware cache. All experiments are run on the "slow" system and the IBM 9LZX disk.*

allocated such that consecutive data blocks do not span track boundaries and accesses 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 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 [15, 40]. 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 (and FFS as well) 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 3 shows that the Traxtent SDS with prefetching results in roughly a 40% improvement in bandwidth for medium-sized files.

Structural Caching: We next discuss the use of semantic information in caching within an SDS. Simple LRU management of a disk cache is likely to duplicate the contents of the file system cache [41, 43], and thereby wastes memory in the storage system. This waste is particularly onerous in storage arrays, due to their large amounts of memory. In contrast, an SDS can use its structural understanding of the file system to cache blocks more intelligently, and thus avoid wasteful replication. We explore the caching of blocks in both volatile memory (DRAM) and non-volatile memory (NVRAM), as each presents unique opportunities for optimization.

We first examine a simple optimization that avoids worst-case LRU behavior. This File-Aware Caching SDS (FAC SDS) exploits knowledge of file size to selectively cache blocks from files that are small enough to fit into the available cache, or that are from files that are not being accessed sequentially. This strategy avoids caching blocks from large files that are being scanned and would otherwise flush the cache of all other blocks.

To implement file-aware caching, the FAC SDS identifies cacheable blocks using indirect classification and association; in this case, the hash table holds block addresses that correspond to those files that meet the caching criteria. As described previously, this may cause the SDS to misclassify blocks in those cases when the file inode is written to disk after the data blocks. The FAC SDS also keeps a small amount of state per active file in order to detect sequential access patterns.

Table 4 shows the performance of the FAC SDS under a database workload. In this scenario, we run TPC-B transactions, and periodically intersperse large file scans into the system, thus emulating a system running mixed interactive and batch transactions. Whereas the large scan flushes the contents of a traditional LRU-managed cache (and hence degrades performance for subsequent transactions), the file-aware cache does not cache blocks from large scans, thus keeping the transactional tables in SDS memory and improving performance.

We next examine how an SDS can use semantic knowledge to store important structures in non-volatile memory. We explore two different possibilities. In the first, we exploit semantic knowledge to store the ext3 journal in NVRAM. To implement the Journal Caching SDS (JC SDS), the SDS must recognize traffic to the journal and redirect it to the NVRAM. Doing so is straightforward, as the EOF tool determines which blocks belong to the journal. Thus, by classifying and then caching data reads

	Create	Create+Sync
ext3	4.64	32.07
+LRU_8 SDS	5.91	11.96
+LRU_{100} SDS	2.39	3.35
+Journal Caching SDS	4.66	6.35

Table 5: **Journal Caching.** *The table shows the time to create 2000 32-KB files, under ext3 without an SDS, with an SDS that performs LRU NVRAM cache management using either 8 MB or 100 MB of cache, and with the Journal Caching SDS storing an 8 MB journal in NVRAM. The* **Create** *benchmark performs a single* `sync` *after all of the files have been created, whereas the* **Create+Sync** *benchmark performs a* `sync` *after each file creation, thus inducing a journaling-intensive workload. These experiments are run on the "slow" system running Linux 2.4 and utilizing IBM 9LZX disk.*

	Create	Read	Delete	PostMark
FFS	73.61	5.14	64.41	230.0
+LRU_8 SDS	1.67	211.10	1.32	333.0
+LRU_{100} SDS	1.67	3.51	4.32	12.0
+MDC SDS	1.76	11.34	0.91	19.0

Table 6: **Meta-data Caching.** *The left three columns of the table show the time in seconds to complete each phase of the LFS microbenchmark [33] (in this experiment, the LFS benchmark creates, reads, and deletes 6500 1-KB files). The right column shows the total time in seconds for the PostMark benchmark, run with 5000 files, 5000 transactions, and 71 directories. The rows compare performance under NetBSD FFS on the "slow" system and IBM disk without an SDS, with an SDS that performs LRU NVRAM cache management using either 8 MB or 100 MB of cache, and with the MDC SDS strategy.*

and writes to the journal file, the SDS can implement the desired functionality.

In the second, we place all of the meta-data (bitmaps, inodes, indirect blocks, and directories) of NetBSD FFS 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 Meta-data Caching SDS (MDC SDS) uses an additional map to record their in-core location.

Tables 5 and 6 show the performance of the JC SDS and the MDC SDS. In both cases, simple NVRAM caching of structures such as a journal or file system meta-data are effective at reducing run times, sometimes dramatically, by greatly reducing the time taken to write blocks to stable storage. An LRU-managed cache can also be effective in this case, but only when the cache is large enough to contain the working set. One of the main benefits of structural caching in NVRAM is that the size of the cached structures is known to the SDS and thus guarantees effective cache utilization. A hybrid may combine the best of both worlds, by storing important structures such as a journal or other meta-data in NVRAM, and managing the rest of available cache space in an LRU fashion.

In the future, we plan to investigate other ways in which semantic information can be used to improve storage-system cache management. For example, an SDS can use certain types of meta-data updates (such as last-accessed-time updates in an inode) in order to ascertain what files are likely to be in the file system cache above. Prefetching within an SDS is also likely to be more intelligent, as the system has file awareness and thus can make a better guess as to which block will next be read. Finally, blocks that have been deleted can be removed from the cache, thus freeing space for other live blocks.

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" [20]. 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 (incorrectly) placed such functionality within the file system by over-writing deleted file blocks multiple times with various patterns [6]. However, this does not guarantee that the data is removed from the disk; other copies of various data blocks may exist, due to bad-block remapping or other storage system optimizations [20, 42]. Further, multiple consecutive file-system writes may not reach the disk media due to NVRAM buffering [5]. An SDS is the only locale where a secure delete can be implemented, since it can ensure that no stray copies of data exist and that over-writes are performed on the disk.

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. Also note that we currently must mount the ext2 file system synchronously for secure deletion to operate correctly; we are investigating techniques to relax this requirement as a part of future work.

Table 7 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-

	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 7: **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 32-KB 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 "slow" system running Linux ext2 mounted synchronously upon the IBM 9LZX disk.*

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

Table 8: **Journaling.** *The table shows the time to complete each phase of the LFS microbenchmark in seconds with 1000 32-KB files. Four different configurations are compared: ext2 on Linux 2.2 mounted synchronously, the same mounted asynchronously, the journaling ext3 under Linux 2.4, and the Journaling SDS under a synchronously mounted ext2 on Linux 2.2. This experiment took place on the "slow" system and the IBM 9LZX disk.*

overwrite until the disk is idle, instead of performing it immediately; freeblock scheduling may further minimize the performance impact [26].

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, the Journaling SDS requires a great deal of precise information about the file system.

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.

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 a 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.

Similar to the Secure-deleting SDS, 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 process. To avoid this problem, the Journaling SDS looks for a suspicious sequence of 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 monitors elapsed time since the last commit; if dependencies prolong the commit by more than a certain time threshold, it suspects an asynchronous mount and aborts.

We evaluate both the correctness and performance of the Journaling SDS. To check correctness, we crashed the file system numerous times, and ran `fsck` to verify that no inconsistencies were reported. The performance of the Journaling SDS is summarized in Table 8. 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 journal-

EOF Fingerprinting			
Probe process	1061		
In SDS	2542		
SDS Infrastructure		**Case Studies**	
Initialization	395	Traxtents	1320
Hash table and cache	2122	File-aware cache	203
Direct classification	195	Journal cache	305
Indirect classification	75	Meta-data cache	235
Association	15	Secure delete	80
Operation inferencing	1105	Journaling	2440

Table 9: **Code Complexity.** *The number of lines of code required to implement various aspects of an SDS are presented. For the "In SDS" component of the EOF tool, there are 2542 lines of code; roughly 1800 of those lines are shared among all file system types; the rest is file-system specific.*

ing 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 have a noticeable impact. Since the purpose of this case study is to demonstrate that an SDS can implement complex functionality, this small overhead is certainly acceptable.

7.2 Complexity Analysis

We briefly explore the complexity of implementing software for an SDS. Table 9 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, the basic cache and hash tables, and the operation inferencing code. Most of the case studies are trivial to implement on top of this base infrastructure; however, the Traxtent SDS and the Journaling SDS require a few thousand lines of code. Thus, we conclude that including this type of functionality within 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 [23]. 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 new innovations behind existing interfaces. Further, storage system manufacturers can now embed optimizations that previously were relegated to the domain of file systems, enabling vendors to compete along axes other than cost and performance.

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. We have shown that the costs associated with reverse-engineering file system structure and behavior are reasonable.

Many challenges remain, including understanding the generality and robustness of semantic inference across a broader range of file systems. Can more sophisticated file systems across a wider range of platforms be probed to reveal their inner workings? Can approximate information be further exploited to implement interesting new functionality? Can more techniques and tools be developed to assure the correct operation of semantic technology? We believe the answer to these questions is yes, but only through further research and experimentation will the final answer be elicited.

Acknowledgments

We would like to thank the members of the WiND research group for their feedback on the ideas presented within this paper. We also would like to thank Keith Smith for his excellent shepherding, and the anonymous reviewers for their thoughtful suggestions, many of which have greatly improved the content of this paper. Finally, we thank the Computer Systems Lab for all of their tireless assistance in providing a terrific environment for computer science research. This work is sponsored by NSF CCR-0092840, CCR-0133456, CCR-0098274, NGS-0103670, ITR-0086044, an IBM Faculty Award, and the Wisconsin Alumni Research Foundation. Timothy E. Denehy is sponsored by an NDSEG Fellowship from the Department of Defense.

References

[1] A. Acharya, M. Uysal, and J. Saltz. Active Disks. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, San Jose, CA, October 1998.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic Function Placement for Data-intensive Cluster Computing. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 307–322, June 2000.

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

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

[5] M. Baker, S. Asami, E. Deprit, J. Ousterhout, and M. Seltzer. Non-Volatile Memory for Fast, Reliable File Systems. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, Boston, Massachusetts, October 12–15, 1992. ACM SIGARCH, SIGOPS, and SIGPLAN.

[6] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *The Tenth USENIX Security Symposium*, Washington, D.C., August 2001.

[7] J. Brown and S. Yamaguchi. Oracle's Hardware Assisted Resilient Data (H.A.R.D.). *Oracle Technical Bulletin (Note 158367.1)*, 2002.

[8] N. C. Burnett, J. Bent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Exploiting Gray-Box Knowledge of Buffer-Cache Contents. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 29–44, Monterey, CA, June 2002.

[9] C. Chao, R. English, D. Jacobson, A. Stepanov, and J. Wilkes. Mime: a high performance parallel storage device with strong recovery guarantees. Technical Report HPL-CSP-92-9rev1, HP Laboratories, November 1992.

[10] C. S. Collberg. Reverse Interpretation + Mutation Analysis = Automatic Retargeting. In *Conference on Programming Language Design and Implementation (PLDI '97)*, Las Vegas, Nevada, June 1997.

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

[12] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, pages 177–190, Monterey, CA, June 2002.

[13] I. Dowse and D. Malone. Recent Filesystem Optimisations on FreeBSD. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

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

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

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

[17] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In *Proceedings of the 8th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII)*, October 1998.

[18] G. A. Gibson, D. F. Nagle, K. Amiri, F. W. Chang, H. Gobioff, E. Riedel, D. Rochberg, and J. Zelenka. Filesystems for Network-Attached Secure Disks. Technical Report CMU-CS-97-118, Carnegie Mellon University, 1997.

[19] J. Gray. Storage Bricks Have Arrived. Invited Talk at the First USENIX Conference on File And Storage Technologies (FAST '02), 2002.

[20] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *The Sixth USENIX Security Symposium*, San Jose, California, July 1996.

[21] R. Hagmann. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, November 1987.

[22] W. C. Hsieh, D. Engler, and G. Back. Reverse-Engineering Instruction Encodings. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '01)*, Boston, Massachusetts, June 2001.

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

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

[25] R. King. Dirty filesystem bug in 2.4.9-21 ext3. https://listman.redhat.com/pipermail/ext3-users/2002-April/003343.html, March 2002.

[26] C. R. Lumb, J. Schindler, and G. R. Ganger. Freeblock Scheduling Outside of Disk Firmware. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002.

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

[28] A. Morton. Data corrupting bug in 2.4.20 ext3. http://www.uwsg.iu.edu/hypermail/linux/kernel/0212.0/0010.html, Dec. 2002.

[29] J. K. Ousterhout. Why Aren't Operating Systems Getting Faster as Fast as Hardware? In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, CA, June 1990.

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

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

[32] E. Riedel, G. Gibson, and C. Faloutsos. Active Storage For Large-Scale Data Mining and Multimedia. In *VLDB*, New York, NY, August 1998.

[33] 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.

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

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

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

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

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

[39] T. Ts'o and S. Tweedie. Future Directions for the Ext2/3 Filesystem. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Monterey, California, June 2002.

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

[41] T. M. Wong and J. Wilkes. My Cache or Yours? Making Storage More Exclusive. In *The Proceedings of the USENIX Annual Technical Conference (USENIX '02)*, Monterey, CA, June 2002.

[42] X. Yu, B. Gum, Y. Chen, R. Y. Wang, K. Li, A. Krishnamurthy, and T. E. Anderson. Trading Capacity for Performance in a Disk Array. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, October 2000.

[43] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, Boston, MA, June 2001.