Semantically-Smart Disk Systems: Past, Present, and Future

Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Lakshmi N. Bairavasundaram, Timothy E. Denehy, Florentina I. Popovici, Vijayan Prabhakaran, Muthian Sivathanu[‡]

Computer Sciences Department
University of Wisconsin, Madison

[‡]*Google, Inc.*
Mountain View, California

Abstract

In this paper we describe research that has been on-going within our group for the past four years on *semantically-smart disk systems*. A semantically-smart system goes beyond typical block-based storage systems by extracting higher-level information from the stream of traffic to disk; doing so enables new and interesting pieces of functionality to be implemented within low-level storage systems. We first describe the development of our efforts over the past four years, highlighting the key technologies needed to build semantically-smart systems as well as the main weaknesses of our approach. We then discuss future directions in the design and implementation of smarter storage systems.

1 Introduction

For the past four years our group has been working on new ways to increase the functionality, performance, reliability, and security of storage systems. Our approach has been consistent throughout: how can we build the storage systems of tomorrow, while living within the constraints of the real world? This notion of "design within constraint" is one of the major research thrusts of our group: too often, in the real world, one must deal with how things are rather than how we want them to be.

In the world of storage, one of the main constraints one encounters is that presented by the interface to storage. Typically, a disk or RAID presents a linear array of blocks to clients; each block can be read or written (SCSI is a good example [21]). This interface has many advantages, primarily in that it is a simple and portable way for file systems and other direct clients of storage to access disk drives. Virtually all of the complexity in head positioning, error handling, and other details of drive access are hidden from the client.

However, such a high-level abstraction has its downsides as well. As Lampson famously said, "Don't hide power" [15]; unfortunately, the array-based interface to storage does just that, preventing a large number of interesting and useful pieces of functionality from being implemented. For example, research suggests that rotationally-aware disk schedulers would greatly improve performance [14, 25]; unfortunately, the low-level information required to perform such scheduling is hidden behind the disk interface. Many other examples exist [8, 16, 23, 32, 33], but all have the same flavor: the desired functionality requires information from both the higher-level system (*e.g.*, the file system) *and* the lower-level system (*e.g.*, the disk).

One natural solution to this problem is simply to change the interface to storage [1, 11]. However, such change is fraught with peril, requiring broad industry consensus and massive upheaval in existing infrastructure.

Hence, we embarked on an alternate course: what would be required to build the storage systems of tomorrow despite the limitations of today's interfaces? Many researchers (including ourselves) have taken the "high road" towards this end, building file systems that have more awareness of the disk system underneath [5, 6, 7, 8, 22, 23, 24, 31]; here, we instead chose the "low road" of enhancing low-level storage systems with knowledge of the file system's above them. One motivating reason for choosing the storage system as the target of innovation was a practical one: the multi-billion dollar storage industry largely builds and ships block-level storage systems.

In this paper, we thus describe our work on *semantically-smart disk systems*. As compared to a typical "dumb" storage device, a semantically-smart system has knowledge of file system data structures and operations, and can use this knowledge to build new and interesting storage systems. Our work has focused on improving the performance [3, 30], reliability [29], and security [27] of storage systems, by applying novel techniques not possible in typical devices.

We present the development of our work over the past four years, discussing key pieces of technology and reflecting on how each step led us down new paths. We then discuss how such technology may filter into the industrial world, and the likely utility of semantically-smart techniques over time. The most surprising aspect of our work was the need for sound theoretical underpinnings; *both* theory and practice are required to build correctly functioning semantically-smart disks.

The rest of the paper is organized as follows. In Sec-

tion 2, we provide background, describing our work in *gray-box* techniques. We then present the first and second generation of semantically-smart disk prototypes, in Sections 3 and 4, respectively. We discuss the importance of a theoretical framework in Section 5, present future directions in Section 6, and conclude in Section 7.

2 Background

Our work in semantically-smart disk systems finds its roots in our earlier research in *gray-box* techniques [2, 4, 7, 17]. The basic idea behind the gray-box approach is quite simple: when you are building a component within a system, you often have a great deal of knowledge of how the other components of that system are designed and implemented. By taking advantage of the knowledge within the component you are building, you are exploiting the fact that these other components are not *black boxes*: rather, their inner-workings are known and hence can be exploited. However, the knowledge that you have of other components is not perfect (*i.e.*, they are not *white boxes*). Hence, dealing with this imperfect knowledge is often a critical challenge in leveraging gray-box techniques.

As an example, consider an application that is going to scan through a set of files. If the application knows whether the underlying operating system has some of these files in its cache, and the application has the flexibility to choose the order of access, it should likely access cached files first. Doing so improves latency (the files in cache are accessed more quickly) as well as bandwidth (if other files were accessed first, they may displace the files in cache and thus the operating system would have to fetch them from disk again).

The problem that arises in our example is that a given operating system may not reveal this information. Thus, the challenge becomes: how can we take advantage of our gray-box knowledge of operating systems (*i.e.*, that they have caches, and that their cache replacement policies are relatively well-known) to determine what the likely contents of the cache are?

Many different approaches are possible. For example, some of our earlier work demonstrated the utility of probing the cache [2]; by accessing a few blocks of a given file, and timing how long it takes to access them, an application can determine whether the file is present within the cache (with high probability). Subsequent work took a different route: by first learning the behavior of the cache-replacement algorithm (*e.g.*, whether it is recency- or frequency-based and whether it uses history to make replacement decisions), an application can then simulate the replacement algorithm and build a reasonably accurate model of the contents of the cache [4].

After this initial line of work targeting the application-OS boundary, we began to consider the interface between file systems and the block-level storage systems beneath them. It soon became clear that the very simple interface to storage, while having many benefits [33], is also quite limited [8, 11]. Many interesting optimizations and enhancements to functionality require information from *both* the file system and the storage system; however, such information is difficult to come by, as neither layer in this "storage stack" has information about the other layer.

One possible solution to this "information gap" is to change the interface to storage; indeed, many people have been advocating such a change for years [1, 11]. However, such change is problematic, for the following reasons. First, the broad industry consensus required to instantiate such a change is enormous; not only would disks and RAIDs have to add new capabilities, but the clients of such systems would also have to migrate to use them before the benefits were obvious (a "chicken-and-egg" problem). Second, changing to a successful new interface requires one to anticipate many possible usage scenarios; however good the design team, it is unlikely that all relevant situations would be taken into account. Finally, such change is expensive; huge investments are likely required to enable them and make them pervasive.

Given the problem presented by the limited interface between file systems and storage, and given that an explicit interface change was unattractive for a variety of reasons, we soon found ourselves wondering: how far could we get if we didn't change the interface? What if, instead, we built "smarter" storage systems, that either learned or inferred information about the file systems above them? In essence, could we obtain many of the benefits of a new interface *without* requiring a change to the storage interface?

3 First-Generation Systems

It was with this mindset that we began work on our first effort about semantically-smart storage systems, published in FAST '03 [30]. This work had three major thrusts: a tool (EOF) that could be used to automatically infer data structures of a client file system, a set of run-time techniques that a disk requires to determine relevant pieces of file system information, and a collection of case studies to demonstrate the utility of semantic-awareness in storage. An additional case study was later published in ISCA'04 [3]. We now discuss these pieces in turn, describing the challenges of each as well as what we learned.

3.1 Offline Techniques: Extracting Static Information with EOF

Our first challenge was a simple one: how would a disk or RAID system gain knowledge of file system data structures? One approach is to simply embed static file system information within the disk or RAID itself; such a system comes built-in with knowledge of the file system data structures and their locations on disk. Although this is

the approach we take in later work, we initially felt such an approach was too limiting and wondered if we could automate the process.

EOF (for "Extraction Of File systems") is a tool that does just that. The basic operation is simple. First, a user-level process on the host issues a series of disk requests to the disk. Then, an in-disk agent monitors the resultant traffic. By carefully controlling the exact file operations that are issued to the file system, EOF can infer a great deal of knowledge about on-disk structures.

The technique that EOF uses is *isolation* combined with *known patterns*. For example, if two blocks get written to disk during a given test, and we know that one is a data block and the other is an inode, we can identify each as follows. First, we can fill the data block with a known pattern; by monitoring the contents of all written blocks, the storage system can detect such data blocks. Then, because the disk knows that for the given workload, only an inode and data block are written to disk, it can successfully isolate the inode block (it is the block that is not filled with a known pattern). In this manner, EOF can acquire a remarkable amount of detailed information about the on-disk structures of the file system.

3.2 On-line Techniques: Classification, Association, and Operation Inferencing

Beyond EOF, we realized that an important component within any semantically-smart disk system was *on-line inference*. Specifically, despite static knowledge, the disk system needs to monitor current traffic to make inferences about the state of the file system.

For example, the disk system may wish to know whether a given block is live or dead. To make such an inference, simply knowing the static location of bitmaps is not enough; rather, one must examine the contents of the bitmap (it is actually more complicated that this, as we will see in Section 4).

Hence, we developed a set of basic on-line techniques that semantically-smart disk systems could use to garner this type of knowledge. The first and most basic is called *direct classification*. With this technique, the disk system examines the block address of a read or write request and uses it to determine the type of the block. For example, if a read or write is directed to the inode region of the disk, simply checking the address is sufficient to determine that the given block contains inodes.

A slightly more sophisticated form of classification is known as *indirect classification*. With this technique, one examines the contents of other blocks to determine the type of a given block. For example, to determine that a block holds directory contents (in a typical UNIX file system), one must examine the inode that points to this block; we call the process of monitoring inode contents *inode snooping*. Indirect blocks can be similarly identified.

A second technique we call *block association*. This technique is used to connect related blocks in a simple and efficient manner. For example, when a data block is read or written, it may be useful to know to which inode the block belongs. A table that maps these associations delivers exactly this information.

A final technique is what we term *operation inferencing*. With this method, a semantically-smart disk can infer when higher-level "operations" are invoked by the file system above. For example, it may be useful to infer file creations or deletions. In a UNIX file system, these operations can be detected by monitoring changes in file system state. For example, by observing a change in an inode bitmap, one can infer the creation or deletion of a file (a bit that was 0 set to 1 indicates a creation, and a bit at 1 becoming 0 indicates deletion).

3.3 Case Studies

With our basic infrastructure and techniques in place, we constructed a set of prototype semantically-smart disks to demonstrate their utility. Most of our prototypes were built as software pseudo-device drivers and mounted beneath real file systems. In a few cases (mentioned below), we utilized simulation to explore the given idea.

The first case study we discuss is an in-disk implementation of track-aligned extents [23, 30]. The basic idea here is to allocate files such that they fit within a track if possible; by avoiding costly track-switches during file access, performance can be improved. Our disk-level implementation used its semantic knowledge of file system structures to influence file system placement to become track-aligned; specifically, by marking blocks that are on track boundaries as allocated, the disk can coerce the file system into allocating files in the proper manner. The resulting performance improvement was noticeable (40%).

Our second case study focuses on caching. This system, known as X-RAY [3], tries to infer the contents of the OS cache by monitoring the stream of traffic the disk generates. The key insight in X-RAY is that every time a file is read, its inode is updated with a new access time and eventually flushed to disk. By watching for inode access-time updates, X-RAY can build a coarse model of what is in the OS cache. Once X-RAY knows what is in the cache above, it can do a better job of managing its own cache, by aiming at exclusivity [34]. Simulations show that performance can be dramatically improved with the smarter second-level caching strategy X-RAY employs.

The final case study we focus on here is the implementation of journaling beneath a non-journaling file system [30]. This turned out to be the most difficult case study to implement. At block level, what the semantically-smart disk tries to infer is when a file system "transaction" is taking place, *e.g.*, when a group of

related updates are occurring. What makes this challenging is file system behavior; because file systems fundamentally delay, reorder, and sometimes filter out operations to disk, the disk has a difficult time decoding exactly what has happened. Our solution to this problem at this point was simple: mount the file system synchronously, thus guaranteeing that all updates are reflected to disk in a complete and timely manner. The result was a disk that implemented journaling with all its associated benefits under a non-journaling file system (in this case, Linux ext2).

3.4 Lessons Learned

Our first year of working on the project thus yielded many interesting results. We saw that we could infer many on-disk structures automatically, through the techniques we developed for EOF. We also developed numerous on-line techniques to determine the true state of the file system and infer which operations it was invoking. Finally, through our case studies, we observed the great potential semantically-smart disk systems had, enabling new and interesting storage functionality, all without change to the file system above.

Our initial work also demonstrated numerous difficulties with our approach. We had originally thought the on-line techniques would be challenging to develop; we soon understood that the asynchronous nature of modern file systems would greatly complicate any on-line inference we wished to perform. Clearly there was more to be understood here. We also came to see that embedding static information about data structures in a disk was probably reasonable; on-disk data structures tend to evolve slowly and there are not too many file systems in the world. Hence, we did not work to improve EOF or the automatic data structure inference tools, instead assuming any semantically-smart would ship with built-in knowledge of important file system structures.

We also were surprised to learn of the difficulties of working underneath the Linux ext2 file system. We chose ext2 because we thought it would be the simplest to operate underneath; instead, it soon proved to be the most challenging. The primary reason for this hardship was the *laissez faire* manner in which ext2 writes blocks to disk: unlike most UNIX-based file systems, ext2 imposes no ordering of any kind on disk writes, making semantic inference quite challenging (as we discuss further in §4).

Our broadest conclusion from this work came from our experience with case studies. It was clear that with each case study, we had learned a lot about the technology needed to build semantically-smart systems. Hence, to develop the technology further, we would need to find interesting pieces of storage functionality to develop in the semantically-smart way. We found ourselves ruminating about the possibilities. We were looking for one such bit of functionality, but we were lucky: we found two.

4 Second-Generation Systems

Our second generation of semantically-smart prototypes took semantically-smart technology to new heights, greatly increasing our own understanding of how such systems could work. We also began to see the limitations of the approach, which we believe was only possible because of the extremes to which we pushed the technology. The primary contribution of this work is understanding how to operate under file systems with asynchronous operations when correctness is required.

This second generation of semantically-smart systems is comprised of two in-depth case studies: D-GRAID, which is a RAID array that degrades gracefully [29], and FADED, a secure-deleting disk that operates under asynchronous file systems [27] (hence removing the major limitation in our earlier attempt at secure delete [30]). We discuss each in turn, and then present the lessons we learned through these two works.

4.1 D-GRAID: Degrading Gracefully

D-GRAID [29] exploits semantic intelligence within a disk array to place file system structures across disks in a fault-contained manner. Thus, when an unexpected failure of second disk occurs [12], D-GRAID continues to operate, serving those files that can still be accessed. There are two key techniques D-GRAID uses to provide this higher level of availability.

The first technique is to *replicate naming and metadata structures* of the file system to a high degree while using standard redundancy techniques for data. Thus, with a small amount of overhead, excess disk failures do not render the entire array unavailable. Instead, the entire directory hierarchy can still be traversed, and only some fraction of files will be missing, proportional to the number of missing disks.

The second technique is *fault-isolated data placement*. To ensure that meaningful units of data are available under failure, D-GRAID places semantically-related blocks (*e.g.*, the blocks of a file) within the storage array's unit of fault-containment (*e.g.*, a disk). By observing the natural failure boundaries found within an array, failures make semantically-related groups of blocks unavailable, leaving the rest of the file system intact. Unfortunately, fault-isolated data placement improves availability at a cost; related blocks are no longer striped across the drives, reducing the parallelism found within most RAID techniques [10]. To remedy this, D-GRAID implements *access-driven diffusion* to improve throughput to frequently-accessed files, by copying the blocks of "hot" files across the drives of the system.

Underneath Linux ext2, determining which blocks are semantically-related is challenging because blocks are dynamically typed (*e.g.*, a block can be a user-data block, an indirect-pointer block, or a directory-data block) and be-

cause the order of writes from the file system to disk can be arbitrary. As a result, the storage system cannot always accurately classify the type of each block. For example, a block B filled with indirect pointers can only be identified as such by observing the corresponding inode, I_B. However, due to the reordering behavior of the file system, it is possible that in the time between the disk writes of the inode and the indirect block, block B was freed from the original inode and was reallocated to another file as a normal data block. The disk cannot know this since the operations took place in memory and were not reflected to disk. Thus, the inference made by the semantic disk can be wrong due to the inherent staleness of the information.

D-GRAID deals with this uncertainty by allowing the fault-isolated placement of a file to be compromised for a limited amount of time. However, this time is bounded, because once the inode of a file is written, D-GRAID will detect the correct classification and move the block accordingly. D-GRAID contains further optimizations to reduce the number of misclassifications by checking that the contents of possible indirect blocks appear valid (*i.e.*, they contain some number of valid unique pointers or null pointers, and only the first so many slots are non-null).

We implemented D-GRAID under both ext2 and VFAT, and overall, D-GRAID behaves as desired. Our analysis shows that D-GRAID allows users to access files when additional disk failures occur within the RAID; with naming and meta-data replication, the percentage of accessible files matches the percentage of working disks. Even better, if we utilize "process availability" as the figure of merit (*i.e.*, the number of processes that run unaffected under disk failure), D-GRAID degrades much better than the expected linear drop-off, because many processes access no user files and therefore run successfully even if most storage is unavailable.

4.2 FADED: Gone and Forgotten

Smarter storage systems need to understand whether blocks are live or dead [32, 35]. We have investigated how block liveness can be inferred within semantically-smart storage; specifically, we have explored the difficult case of how to infer *generational liveness*, that is, whether a block currently belongs to a given live file. In this context, we implemented FADED (A File-Aware Data-Erasing Disk), which implements *secure delete*, ensuring that deleted data cannot be recovered from the disk [28]. Secure delete functionality pushes on the disk's ability to perform correct inferences: a false positive in detecting a delete leads to irrevocable deletion of valid data, while a false negative results in deleted data being recoverable.

When FADED detects a file is deleted, FADED *shreds* all of the blocks belonging to that file by overwriting each block multiple times with specific patterns. The fact that a block should be shredded can be detected in different ways: FADED may see that the corresponding bit in the bitmap is cleared (indicating the block has been freed), the generation count in an inode is incremented (indicating the inode has been freed and reallocated), or the block is pointed to by a different inode (indicating the block has been freed and then reallocated to a different file).

The challenge we address is that, again given reordering and reuse in the file system, when a block is pointed to by a different inode, FADED cannot definitively know whether the current contents of the block are those for the new or the old file. FADED deal with such uncertainty by being conservative and converting an apparent correctness problem into a performance problem (*i.e.*, FADED may perform more shredding operations than required). The mechanism we introduce is that of a *conservative overwrite*, which erases past layers of data on the block, but leaves the current contents of the block intact. Using conservative overwrites means that valid data can never be inadvertently shredded, but it also has an associated overhead: certain suspicious blocks need to be tracked and shredded multiple times.

In our prototype implementation, we found that two minor changes were needed in ext2 to operate correctly on top of FADED: the first ensures that file truncates are treated as deletes; the second ensures that our inability to definitively classify indirect blocks does not lead to missed deletes. When using FADED under a typical UNIX workload, we find that the implicit inferences and conservative overwrites impose approximately a 10% overhead compared to a disk with perfect information.

4.3 Lessons Learned

By implementing these two challenging case studies, we learned a great deal about semantically-smart disk systems and the fundamental challenges they pose to system designers. The most important lesson was that living with uncertainty is at the core of building such systems; due to the asynchronous nature of file systems, in the worst case the disk system receives incomplete information regarding the state of the file system at a given time.

We also learned that despite this imprecision, interesting prototypes can still be constructed. Through careful design, both D-GRAID and FADED worked around the lack of complete information and achieved their goals. However, in many cases subtle reasoning was required in order to build robust working prototypes that handled all corner cases. Indeed, many times we were deep into an implementation and only then realized a problem with our approach, requiring us to go back to the drawing board and rethink what we were doing. The more we did this, the more we realized that we needed more than just "being careful"; what we needed was a theory of how file systems and disks interacted.

5 Beyond Systems: Some Theory

We thus began an effort to build a more formal logic of file system and disk interactions [26]. Although this logic began as a means for reasoning about semantically-smart disks, we soon realized that the possibilities were much broader; indeed, such a logic could be used by file system developers as well, to better understand the complex interactions between file systems and disks.

The logic begins with a set of basic entities: *containers*, *pointers*, and *generations*. A file system is simply a collection of containers, linked via pointers. When a container is reused (*i.e.*, freed and then used again), it represents a new generation.

The logic is then formulated through *beliefs* and *actions*. A belief is used to model the state of the file system, either on-disk or in-memory, and an action changes the state of the file system (and hence which beliefs are true at a given time). Fundamental to understanding the impact of actions on beliefs is the *ordering* among the actions, and hence special care must be taken in constructing the temporal relationship between actions. Proofs are finally constructed by starting with basic axioms and applying a series of *event sequence substitutions*; for example, if (α happens before β) implies γ, then wherever we observe that (α happens before β), we can simply replace this subsequence with γ.

Some of our initial results are as follows. First, we prove the correctness of existing file system consistency-maintenance techniques such as soft updates [9]. Further, we also show how the Linux ext3 file system is needlessly conservative in how it performs transaction commit, demonstrating how the logic can be used to enable aggressive performance optimizations. We show how the logic can aid in the development of new functionality, by building and analyzing the correctness of a *consistent undelete* functionality in Linux.

Overall, we found that even a simple logical framework such as ours was critical in the development of semantic technology. Wherever reasoning about disk interaction is required, we believe that a more formal approach is required to build robust and correct systems.

6 Future Directions

Throughout the semantic disks project, we learned a great deal about file systems, disk systems, and their interactions. We now harness that experience to look forward and ruminate on the possible future of semantic disk technology in block-level storage and beyond.

One primary question regarding semantic techniques is their applicability in the "real world". As some of our case studies are quite complex, it seems unlikely that an industry that must fundamentally be conservative will adopt our approach. Therefore we think that successful industry adoption will be aimed at less radical case studies. For example, imagine a disk array that performed smarter prefetching by paying attention to file boundaries. Although this too requires semantic knowledge, it does not require much, and if it is wrong, only (perhaps) performance will suffer.

Another question is whether semantic inference can be applied to other clients of disk systems, such as database management systems. We have already performed some initial work along these lines [28], and have met with mixed success; while some techniques translate readily, the more complex and specific data structures of a typical DBMS do complicate matters occasionally. However, some DBMS structures are ripe for the kind of reverse engineering we advocate; in particular the transaction log is replete with information about what the DBMS is currently doing and hence a likely candidate for future semantic technology.

Along these lines, we have also noticed that the sea change in modern file systems towards journaling is likely to make semantic inference easier rather than more difficult. As with a DBMS, a file system journal takes the chaotic update sequence possible with a simple file system such as ext2 and turns it into an orderly and hence more understandable affair. Linux ext2 was perfect as a file system to study semantically-smart disks underneath, as it pushed us to deal with its extreme asynchrony and arbitrary ordering of writes; future systems, if they are able to interpret log contents, will likely be simpler and more easily verified as correct.

One major change to the storage interface, towards object-based disks, may also be on the horizon [1]; with such change, will the need for semantic inference be obviated? After all, these drives generally have more information about how they are being used by clients than typical block-based disks; for example, with a straight one-to-one file-to-object mapping, the drive can easily determine which blocks are currently free. However, even in this evolved interface, we believe there is much room for further inference and semantic technology. For example, directory structure is not a part of the interface, and journaling file systems and databases will still place logs on disk; these structures and many others still require semantic inference to become valuable sources of information for storage systems.

Finally, we believe that there is a broader place for semantic inference technology than simply building better storage systems. Some current work of ours explores using low-level tracing and fault injection to better understand file system performance [13, 18] and failure characteristics [19, 20]. As systems grow increasingly complex, tools to deconstruct their behavior will likely become an integral part of the design, implementation, and maintenance of said systems.

7 Conclusions

We have presented a retrospective of our work on semantically-smart disk systems. This work began with a simple question ("how smart can we make block-level disks without changing the disk interface?") and evolved into the development of a series of increasingly challenging case studies and the beginnings of a more formal theory for understanding file system and disk interactions. In our modern world, avoiding the constraints placed upon us by layering and other system structuring artifacts is nearly impossible; with semantic inference, however, we believe we have provided a means to reclaim some of what is lost to the nature of such designs.

References

[1] D. Anderson. OSD Drives. www.snia.org/events/past/developer2005/0507_v1_DBA_SNIA_OSD.pdf, 2005.

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

[3] L. N. Bairavasundaram, M. Sivathanu, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA '04)*, pages 176–187, Munich, Germany, June 2004.

[4] 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, California, June 2002.

[5] 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, California, June 2002.

[6] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 87–100, San Francisco, California, December 2005.

[7] T. E. Denehy, J. Bent, F. I. Popovici, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Deconstructing Storage Arrays. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*, pages 59–71, Boston, Massachusetts, October 2004.

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

[9] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, pages 49–60, Monterey, California, November 1994.

[10] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume I, pages 40–49, 1993.

[11] 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.

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

[13] H. S. Gunawi, N. Agrawal, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Schindler. Deconstructing Commodity Storage Clusters. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pages 60–73, Madison, Wisconsin, June 2005.

[14] D. M. Jacobson and J. Wilkes. Disk Scheduling Algorithms Based on Rotational Position. Technical Report HPL-CSP-91-7, Hewlett Packard Laboratories, 1991.

[15] B. W. Lampson. Hints for Computer System Design. In *Proceedings of the 9th ACM Symposium on Operating System Principles (SOSP '83)*, pages 33–48, Bretton Woods, New Hampshire, October 1983.

[16] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards Higher Disk Head Utilization: Extracting "Free" Bandwidth From Busy Disk Drives. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, pages 87–102, San Diego, California, October 2000.

[17] J. Nugent, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Controlling your PLACE in the File System with Gray-box Techniques. In *Proceedings of the USENIX Annual Technical Conference (USENIX '03)*, pages 311–324, San Antonio, Texas, June 2003.

[18] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '05)*, pages 105–120, Anaheim, California, April 2005.

[19] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, pages 802–811, Yokohama, Japan, June 2005.

[20] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[21] P. M. Ridge and G. Field. *The Book of SCSI 2/E*. No Starch, June 2000.

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

[23] 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 1st USENIX Symposium on File and Storage Technologies (FAST '02)*, Monterey, California, January 2002.

[24] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, California, April 2004.

[25] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '90)*, pages 313–324, Washington, D.C, January 1990.

[26] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A Logic of File Systems. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 1–15, San Francisco, California, December 2005.

[27] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or Death at Block Level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 379–394, San Francisco, California, December 2004.

[28] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-Aware Semantically-Smart Storage. In *Proceedings of the 4th USENIX Symposium on File and Storage Technologies (FAST '05)*, pages 239–252, San Francisco, California, December 2005.

[29] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, pages 15–30, San Francisco, California, April 2004.

[30] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, California, April 2003.

[31] 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.

[32] R. Wang, T. E. Anderson, and D. A. Patterson. Virtual Log-Based File Systems for a Programmable Disk. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

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

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

[35] 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 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.