Coerced Cache Eviction and Discreet Mode Journaling: Dealing with Misbehaving Disks

Abhishek Rajimwale†, Vijay Chidambaram, Deepak Ramamurthi,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
†*Data Domain, Inc.*
Computer Sciences Department, University of Wisconsin–Madison
abhishek.rajimwale@datadomain.com, {vijayc, scdeepak, andrea, remzi}@cs.wisc.edu

***Abstract*—We present Coerced Cache Eviction (CCE), a new method to force writes to disk in the presence of a disk cache that does not properly obey write-cache configuration or flush requests. We demonstrate the utility of CCE by building a new journaling mode within the Linux ext3 file system. When mounted in this *discreet mode*, ext3 uses CCEs to ensure that writes are properly ordered and thus maintains file system integrity despite the presence of an improperly behaving disk. We show that discreet mode journaling operates with acceptable overheads for most workloads.**

***Keywords*-file systems; disks; journaling; reliability.**

I. Introduction

Fierce competition among vendors and the inexorable progress of technology have moved the computer industry forward in leaps and bounds over the past decades. A case in point is found in the disk industry, where technical feats and cut-throat business practices are commonplace, resulting in rapid innovation over an incredibly short period of time [10]. The results of decades of progress in storage technology are indeed truly remarkable: terabyte-sized disks capable of delivering hundreds of megabytes per second are available for a few hundred dollars or less.

However, progress and competition in storage are not inculpable: a "dark side" of the relentless pressures of competition emerges when one examines the disk industry more closely. Consider disk reliability; in an industry where "pennies matter" [1], [24], drive vendors in the low-end disk market routinely cut corners and reliability features are often the first to go [19]. The result is almost predictable: drives lose or corrupt blocks often enough [6], [7], [24] that high-end RAID manufacturers must include numerous detection and recovery mechanisms [11], [23], [35].

One aspect of disks that all modern systems implicitly rely upon is *write ordering*. Ordering of writes is an essential component of any modern file system [13], as it is required to implement journaling [16], [38], copy-on-write [28], [18], [8], or soft updates [13], [12].

Should systems trust disks to order writes correctly? The high complexity of disk caches, combined with fierce competition among manufacturers, have inevitably led to less-than-perfect implementations [35], [14]. Furthermore, anecdotal evidence from experts both in the file system industry [2] and disk industry [3] suggests that some manufacturers, in an effort to boost performance, explicitly ignore requests to force writes to disk, keeping blocks in cache and eventually writing them to disk in the background.

If disk write ordering cannot be trusted, the file system is now left with the unsavory question: how can it correctly implement an update protocol such as journaling or copy-on-write? In this paper, we explore methods aimed at answering this question.

In particular, we introduce *coerced cache eviction (CCE)*, a new method to flush writes to the disk surface despite an untrustworthy disk cache. We have designed a simple microbenchmark to fingerprint the behavior of a disk cache; the resulting fingerprint gives the information needed for CCE to coerce the disk to flush its cache both thoroughly and efficiently. We have fingerprinted the cache behavior of nine SATA disk drives and have derived the corresponding CCE parameters.

We demonstrate the utility of CCE by implementing a new journaling mode for the Linux ext3 file system. Known as *discreet mode*, when mounted as such the file system issues the CCE whenever it requires ordering between groups of writes, and thus ensures that its write-ahead logging strategy operates as desired. We show that discreet mode ext3 generally operates with low overhead; the exception arises when an application repeatedly calls `fsync()` to force small amounts of user data to disk; in this case, we recommend the use of more efficient "group commit" strategies.

The contributions of this paper are as follows:

- The first exploration of the write-ordering problem, including how file systems should be built if full trust in write ordering does not exist (§II).
- The introduction of a new method to help control write ordering, coerced-cache eviction, even in the presence of misbehaving disks (§III).
- A study of cache flushing behavior of nine modern and diverse disks (§III).

- The development of a new journaling mode, discreet journaling, which uses the CCE primitive to ensure the correct update sequence despite disk misbehavior (§IV).
- A detailed study of the performance of discreet journaling, showing it adds low overheads for most workloads (§V).

We then conclude the paper in §VI.

II. Motivation

In this section, we discuss the problem that some commercial hard drives do not fully conform to the interface specification. We first describe what clients demand of disks, how disks meet the performance demands, and why those performance demands naturally lead to disks that may not properly order writes.

A. What Do Clients Want From Disks?

An ideal disk tries to deliver two opposing demands from clients such as file systems and databases. The first demand is for *durability*: the ability to make information persistent. The second is for *high performance*: making disk operations complete as quickly as possible. Unfortunately, these demands are often at odds.

Durability begins when a block is written to disk; the client, of course, expects to be able to later read back the written data even after a power loss. Hence, the primary usage of disks are for the long-term storage of information. Related to durability is the notion of *write ordering*; methods for updating disks consistently despite the presence of crashes (*e.g.*, journaling [16], copy-on-write [8], [18], [28], and soft updates [13]) require that some writes be made durable before others.

Performance expectations from disk drives begin with the client's desire to have operations complete quickly. The largest burden for performance is due to the intrinsic nature of storage based on rotating media: it is simply quite expensive to read or write data from the disk's surface. Thus, given the slow nature of the magnetic drive, how can the drive improve performance while maintaining durability and write ordering?

B. How Do Disks Improve Performance?

To meet performance demands, disk vendors have developed a variety of techniques over the years. Modern drives allow clients to submit multiple requests at a time, giving the drive the ability to use low-level internal knowledge of layout and positioning to schedule requests in a more optimal manner [20], [33]. The industry has also worked to slowly reduce positioning and seek times, and for high-end disks, positioning takes just a few milliseconds.

The most effective method for a drive to appear fast is not to access the media at all; this is accomplished via the drive cache. For reads, drive caches are helpful as the drive can aggressively prefetch an entire track (or more) in anticipation of future requests.

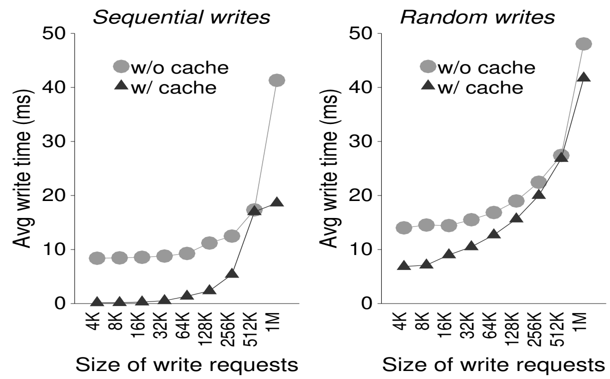

Figure 1. **Disk performance with and without write caching.** *The graph shows the average times in milliseconds taken by the disk for different sized blocks. The top line shows the times taken when the drive cache was turned off and the bottom line shows the times taken when the drive cache was turned on. Tests performed on a HDS7280S 80GB SATA drive.*

Drive caches can also greatly improve performance for writes, as a drive can immediately acknowledge the write's completion. To illustrate these benefits Figure 1 compares the average latency of write requests when caching is enabled versus disabled. For this Hitachi drive, the average write time of both random and sequential requests is significantly longer when write caching is disabled.

Unfortunately, write caching leads to well-known problems with durability and write ordering. First, immediate durability is no longer guaranteed because the disk could fail after it has reported that the write completed, but before the write reached the actual media. As a result, the write is lost.

Second, and more importantly, with write caching, control over ordering is lost. When the drive later destages the cached blocks to the disk media, it may reorder writes to minimize disk arm movement. Thus, the ordering desired by the client is lost.

C. How To Control Ordering Despite Caching?

We are left with a problem: how can a client, such as a file system, control disk write ordering despite the presence of a write cache? One common approach is to disable write-back caching. When a write completes, the file system is guaranteed that the contents of that write have been permanently written to the disk media. Ordering is thus achieved by issuing a write, waiting for its completion, and then issuing the subsequent write.

An alternate approach is to enable the write cache, but allow clients such as file systems to explicitly flush the contents of the cache to disk when ordering is required. For example, the SATA interface specification [37] contains the "flush cache" command.

D. How Do Misordered Writes Occur?

The increasing complexity of disk caches and the demanding development schedules of competitive disk manufactures have led to many bugs in the disk cache software layer. These bugs have been reported in a number of industry papers [14], [35], [39].

More insidiously, the performance benefits of write caching are so high that apparently some manufacturers have decided that write-back caching should be enabled at all times, despite the entreaties of the file system above. Although it is challenging to provide "hard" evidence of this problem, information found in man pages and user discussions on public forums suggests that certain commodity disk drives do ignore flush requests [25], [26], [27]. Evidence from experts within the industry hints at such issues [2], [3]; even the `fcntl` manual page for Mac OSX includes the following interesting tidbit:

F_FULLFSYNC: Does the same thing as fsync(2) then asks the drive to flush all buffered data to the permanent storage device (arg is ignored). This is currently implemented on HFS, MS-DOS (FAT), and Universal Disk Format (UDF) file systems. ... Certain FireWire drives have also been known to ignore the request to flush their buffered data.

Simply put, sometimes disks make mistakes (and thus accidentally forget to write a block to disk). Worse, sometimes disks ignore all flush requests and write the block to disk, but long after acknowledging said write.

E. Are Misbehaving Disks Malicious?

Our model of a misbehaving disk assumes that it may contain one or both of the following two deficiencies. The first way in which a disk may misbehave is that it may simply always cache writes, even when disk caching has been explicitly disabled. The second way in which a disk may misbehave is that it may ignore the command to flush the cache.

It is important to note that these disks are not malicious; that is, they will eventually write data to the media, just not at the time requested. This could happen because of two reasons. First, there could be a *bug* in the firmware. Firmware code consists of hundreds of thousands of lines of low-level concurrent code [24] and thus is prone to errors. Second, there are *incentives* for drive manufacturers to increase performance despite lowered reliability in the face of system crashes. Performance sells drives; reliability is much harder to measure or market. As Kahan famously said about floating-point units: "The fast drives out the slow, even if the fast is wrong." [17].

F. What Are The Consequences of Misordering?

We now discuss the problem of using a disk that misbehaves with respect to flushing its cache, and its

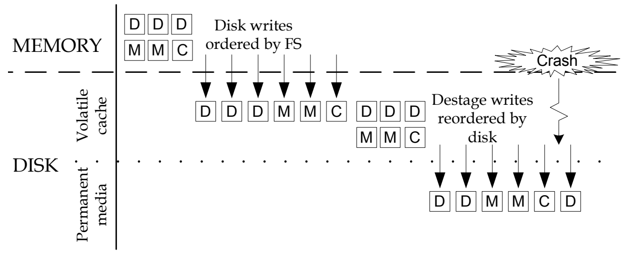

Figure 2. **Problems During Journaling.** *The figure shows ext3 in ordered journal mode. In the figure, 'D' denotes data blocks, 'M' denotes metablocks, and 'C' denotes journal commit blocks. The file system wishes to write data, then metadata, then a commit block to disk. However, because of write caching, the disk flushes the blocks to disk in a different order; thus, when a crash occurs, the commit block has reached the disk but not all metadata, which can corrupt the file system and prevent proper recovery.*

implication on durability and ordering. When disks misbehave, the file system has no guarantee of a particular ordering of blocks written to the disk media. Even if the file system waits for writes to complete and issues barriers to confirm write completion, a misbehaving disk could continue to cache the blocks and reorder them during destaging.

Figure 2 shows the problem of re-ordering for a journaling file system. A crash during background destaging could lead to unpredictable write ordering of blocks, which breaks the data consistency and recovery guarantees provided by the file system. Lack of ordering control can lead to any number of problems, including metadata inconsistency (even after recovery), garbage data in a file, and even unmountable file systems.

Beyond journaling, ordering of writes is required for any file system that tries to maintain on-disk consistency as it runs (as opposed to ext2, for example, which writes data and meta-data to disk in random order, and then runs `fsck` to fix some of the problems that arise). A copy-on-write file system such as Sun's ZFS [8] first writes all new metadata and data to a new location on disk, and then updates the uberblock at the root of the file system tree to point to all current blocks plus the new blocks. If the uberblock reaches disk before the other data, the file system tree will contain inconsistent metadata or garbage or both. Soft updating file systems are no different, and indeed are based around the concept of the careful ordering of writes [13].

It should now be clear: ordering is a requirement if one wishes to use a modern file system. Unfortunately, market pressures and the perfect opportunity to implement a faster but "less correct" disk have led to a reality where despite their best efforts, file systems are unable to control how blocks are written to disk.

III. Coerced Cached Eviction

In this section, we introduce our technique for dealing with misbehaving disks. Our approach of Coerced Cached Eviction (CCE) ensures that the cache has truly been flushed at requested points and thus restores the desired properties of durability and control over ordering of requests.

A. General Approach

The basic idea of Coerced Cached Eviction (CCE) is to generate a *flush* workload of requests to the disk; this flush workload is constructed such that it will replace some set of the current contents of the disk cache, forcing those items to be written back to the disk surface as desired. We refer to the particular set of blocks that the flush workload is attempting to evict from the disk cache as the *target* blocks.

With the CCE mechanism, the file system can ensure that a write request `A` is durably updated to disk before another write request `B`, by first writing `A`, then performing a CCE, and then writing request `B`. In this way, the file system is assured that request `A` has reached the disk surface before request B is issued.

The ideal flush workload has two high-level properties: it has a high probability of actually flushing the target request and it induces negligible performance overhead. As one might expect, a tension exists between these two goals. In general, with more flush traffic, the likelihood of evicting the target increases, but the performance overhead increases as well. Constructing a flush workload that balances these two goals requires understanding not only the running workload but the disk cache algorithm as well.

In the ideal case for performance, the flush traffic is a part of the original workload itself. For example, if independent requests `C` can be found in the workload that do not depend on the ordering of `A` or `B`, then requests from `C` can flush `A` from the disk cache. However, as a first step in this paper, we consider the case where artificial new traffic is added to perform the flush of the target blocks.

Flushing the disk cache with new traffic negatively impacts performance at two points. First, the flush workload may cause blocks other than the target blocks to be evicted from the cache. Second, the flush blocks must be written to disk when they are replaced by other items in the cache. To minimize the time to write those blocks to disk, the flush workload should be small and spatially close to other traffic to the disk. Thus, the most appropriate flush workload depends on a number of factors related to the running workload and the underlying disk, in particular its cache.

B. Understanding Disk Cache Behavior

While a plethora of micro-benchmarks [31], [32], [36], [40] and models [22], [29] exist for revealing and describing the behavior of disks, disk caches are not as well characterized in the literature [21], [34]. Modern disk caches are non-trivial and the parameters are not fully described by manufacturer's specification sheets.

For example, although manufacturers may report the size of the disk cache, disk caches are typically partitioned into read and write portions. A correct and efficient implementation of CCE must know the size of the cache devoted to writes, which is not usually reported. Modern disk caches are also multi-segmented; thus, an ideal CCE would generate a flush workload that only evicts the segments actively containing the target blocks instead of all segments. Unfortunately, even basic information such as the size of each segment and the number of segments is not available, much less information about the replacement policy. Finally, not all write requests may be cached, instead being written through directly to the surface.

Similar in spirit to other fingerprints of the memory hierarchy [5], [9], [30] we have developed an off-line micro-benchmark to create an *eviction fingerprint* for a particular disk. The eviction fingerprint visually characterizes how different requests impact the contents of the cache. We also create a corresponding *performance fingerprint* that shows the time required to perform the flush workload. Given this characterization, one can then determine the flush workload giving the highest probability of success and the least performance overhead for a given disk.

Our flush micro-benchmark operates a number of trials as follows. In a trial, we first perform a write to our target block: this is a single 512-byte sector at a random location on the disk. We then generate a specific flush workload investigating the effect of three parameters: sequential versus random requests, the number of write requests, and the total amount of data in the flush workload; the size of each individual write is calculated as the total data amount divided by the number of requests. To finish, we call `fsync` to ensure that writes are issued to the disk.

We automatically determine whether or not each workload evicted the target block by reading back the target block and measuring the amount of time it required. We assume that a "fast" response (*e.g.*, less than a few milliseconds) means the target is still in the disk cache; a "slow" response means the target has been written back to the disk media. Before the read, we first sleep for one second to increase the likelihood that the disk has completed the preceding write requests and that the read can be serviced immediately. We then repeat these steps for all flush workloads for 50 trials.

Manufacturer	Model	Capacity (GB)	Cache (MB)	Price
Hitachi	HDS7280SASUN80G	80	8	$43
Hitachi	Deskstar OS00163	1024	32	$80
Samsung	Spinpoint M7 HM250HI	250	8	$55
Samsung	Spinpoint F3 HD253GJ	250	16	$45
Western Digital	WD3200AAKS	320	16	$48
Western Digital	WD8000AARS	800	64	$70
Seagate	Barracuda ST3250318AS	250	8	$40
Seagate	Barracuda ST3320613AS	320	16	$50
Seagate	Barracuda ST3750528AS	750	32	$65

Table I : **SATA Disk Drives.** *The table shows the nine SATA disk drives whose caching behavior we have evaluated. The disks were chosen to cover a variety of manufacturers, disk capacities, and cache capacities. Cost represents list price as of April, 2010.*

To evaluate the effectiveness of each flush workload, we calculate the percentage of trials in which the flush workload successfully evicted the target block. We measure the time to issue the writes as the performance of the flush workload.

Our current approach is not without limitations. We do not investigate the sensitivity of cache evictions to frequency of accesses. However, if deemed important, such a modification would not be hard to make.

C. Experimental Results

We have evaluated the effectiveness of various flush workloads for nine commodity SATA disk drives shown in Table I. We selected SATA drives since this is the class of drives reported to most likely misbehave about flushing cache contents (however, we have no indication that any of the disks in our sample are problematic). These disk drives were selected to represent a range of low-cost disks from different manufacturers and with different disk cache capacities (from 8 MB to 64 MB).

1) Eviction Fingerprints: The eviction fingerprints for the nine disk drives are shown in Table II. Each fingerprint shows, for either sequential or random requests, the likelihood of flushing the target block as a function of the number of writes and the total amount of data summed across all writes in the flush workload. For these disks, the number of write requests is varied from one request up to 2560 requests; the total amount of data is varied from 1 MB up to 128 MB.

The dark regions indicate workloads that successfully flush the cache while light regions indicate workloads that do not. The high-level purpose of these fingerprints is to enable us to choose the best flush workload for a particular disk; however, the fingerprints do also reveal some more detailed characteristics of the underlying caches, which we briefly comment on.

First, we note that the cache behavior of disks from the same manufacturer is qualitatively similar across their different models. From this, we infer that a manufacturer is likely to use similar cache structures and replacement algorithms for disk models that are related but vary in capacity. For example, the Hitachi fingerprints have nearly identical structures, but the scales are different by a factor of four for an 8 MB cache versus a 32 MB cache.

Second, whether writes are sequential or random appears to greatly impact the effectiveness of the flush workload. This difference is most dramatic for the Seagate disks, in which sequential workloads are completely ineffectual at flushing the cache regardless of how much data is written. We believe the Seagate disk identifies large sequential streams and chooses not to cache them (and small sequential streams have only a small chance of evicting the target block).

Third, vertical bands of different shades indicate that space in the cache is allocated on a per-write basis; specifically, this hints that the underlying cache is segmented and that a single write regardless of its size is allocated to a single segment.

Horizontal bands of different shades indicate that space in the cache is allocated based on the amount of data written, regardless of the number of individual writes. For example, this effect is exhibited by the Hitachi drives and is most apparent with sequential writes. For these drives, one can flush the cache with one relatively large write.

Finally, some of the flush fingerprints have distinct transition points separating the case where the target block is extremely unlikely to be flushed and the case where the target block is extremely likely to be flushed. For example, for the Hitachi disks with 8 MB caches, sequential writes of less than 2.3 MB have little chance of flushing the target block whereas sequential writes of more than 2.3 MB flush the target block with nearly 100% success. We suspect that these disks use LRU replacement; with LRU, the target block is evicted as long as the flush workload writes enough data.

However, other fingerprints appear much more irregular (e.g., Samsung). We suspect that these disks use a random replacement policy; thus, sometimes the flush workload is allocated to the target segment and sometimes not. Increasing the size of the flush workload increases the chances that one replaces the target block but never guarantees it. Many writes are thus required to even probabilistically evict every target block.

2) Performance Fingerprints: The performance fingerprints for the nine disk drives are shown in Table III. Each fingerprint shows, for either sequential or random requests, the cost of performing a flush with the stated number of writes and total amount of data for that particular disk. The dark regions indicate costly workloads (taking over 500 ms) while the lightest regions indicate fast workloads (taking less than 10 ms).

As expected, issuing a random flush workload is significantly more expensive than a sequential workload

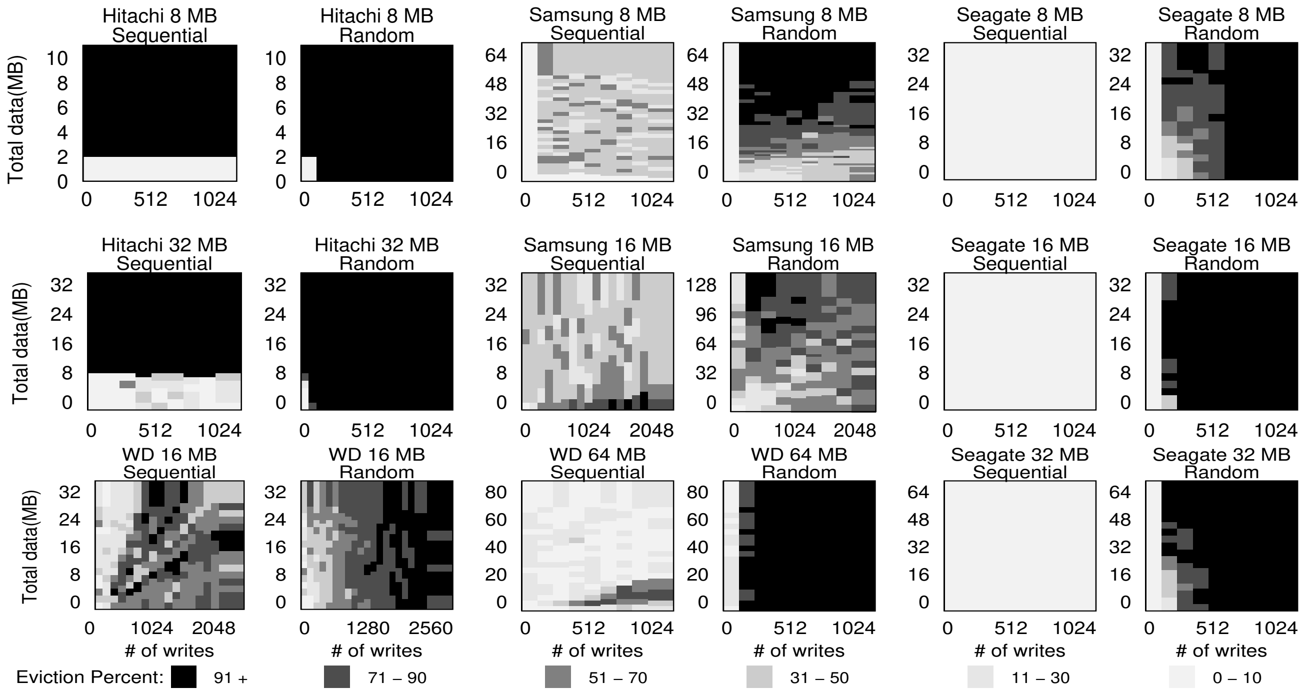

Table II : **Eviction Fingerprints.** *Eviction Fingerprints for SATA Disk drives from different manufacturers.*

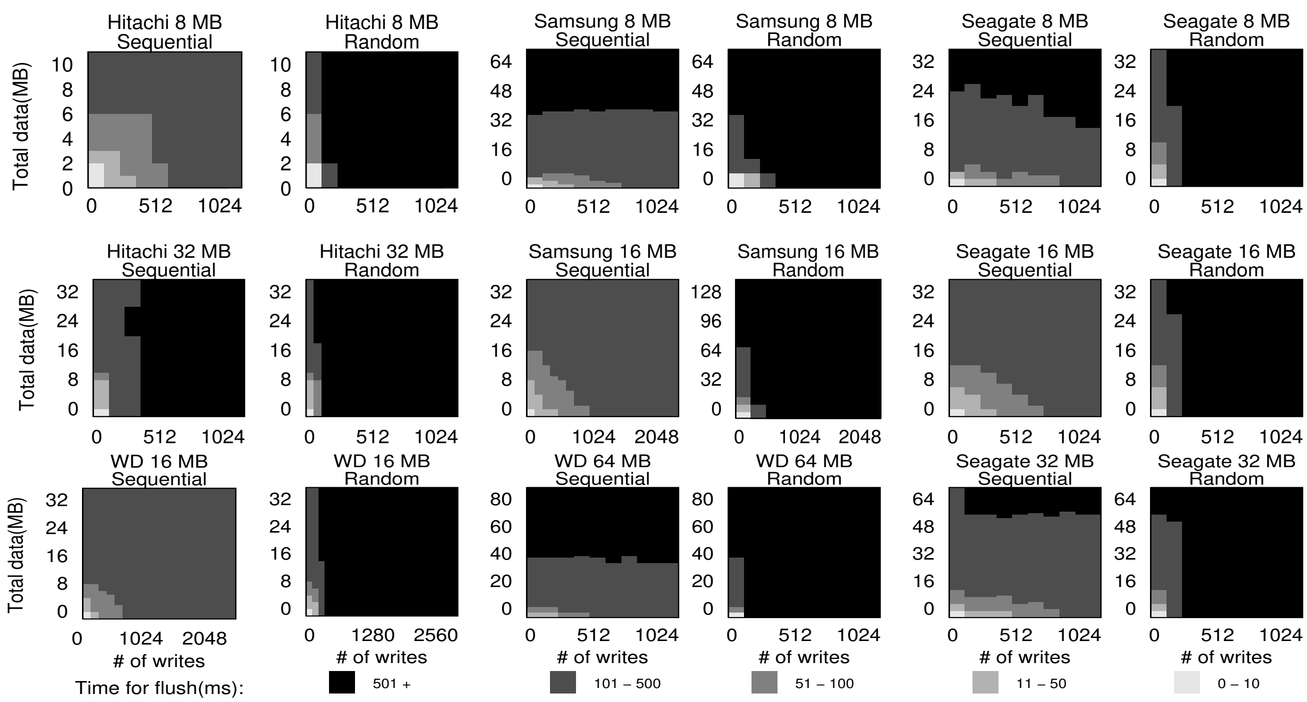

Table III : **Performance Fingerprints.** *Performance Fingerprints for SATA disk drives from different manufacturers.*

Manufacturer	Cache (MB)	Number Writes	Data (MB)	Time (ms)
Hitachi	8	1	2.38	50
Hitachi	32	1	11	87
SAMSUNG	8	128	49	1328
SAMSUNG	16	512	128	2872
Western Digital	16	1792	19	5107
Western Digital	64	256	1	7705
Seagate	8	256	31	870
Seagate	16	128	17	342
Seagate	32	128	37	396

Table IV : **CCE Flush Workloads.** *The table shows the recommended flush workloads for each of the nine fingerprinted disks. We select the flush workload as the one with the best performance and a 100% probability of success in our microbenchmark. The CCE performs the stated number of random write operations with the stated total amount of data. The last column shows the time in milliseconds to perform the CCE.*

due to the additional disk seeks and rotations incurred by the random writes. Across graphs, one can also compare performance across disks.

3) Flush Workloads: From the eviction and performance fingerprints, we can determine the best flush workload for a particular disk drive. The best workload is the one that most effectively flushes the target block from the disk cache with the lowest performance overhead. We note that there exists a trade-off between these two goals. In some cases it will be impossible for the CCE operation to guarantee that a target block has been evicted from the cache; thus, our goal is only to significantly increase the likelihood that writes are sent to the disk media in the correct order in case of a crash. In general, we search for the flush workload with the best performance that has a 100% probability of success in our microbenchmarks.

Table IV shows the flush workload we have derived by combining the eviction and performance fingerprints. On the two Hitachi disk drives it is relatively simple and inexpensive to perform a CCE; one can simply perform a single write of either 2.3 MB (8 MB cache) or 11 MB (32 MB cache) requiring either about 50 ms or 87 ms.

Safely flushing the caches of the other disk drives is more expensive, requiring more random write operations (at least 128) and often more total data (up to 128 MB). Initiating more random writes has a steep performance cost, raising the cost of the CCE up to hundreds of milliseconds for the Seagates, and higher for the Samsungs and Western Digitals.

D. Summary

The success of the CCE operation depends on the disk characteristics. For disks with complex replacement policies in the disk cache, it is difficult to evict a block with certainty; CCE only increases the probability of eviction, and the cost of the flush is directly proportional to the probability of eviction.

IV. Journaling in Discreet Mode

By using the CCE primitive, existing file systems can guarantee data consistency and provide crash recovery even in the presence of misbehaving disks. It can be incorporated into any file system that requires a specific ordering of write requests for correctness. For example, a file system based on soft updates, copy-on-write, or ordered synchronous writes could be modified to use this operation.

In this section, we describe our experience incorporating CCE into a journaling file system, specifically Linux ext3 for both ordered and data journaling modes. We call our new extension **discreet** mode. We recommend that this mode of ext3 be used when one suspects that the underlying disk is not entirely trustworthy.

A. Overview

Journaling in discreet mode does what it should do with hardware that misbehaves – show discretion. That is, this mode of the file system acts in such a manner as to protect itself and its interests without blaming the underlying disk. Journaling in discreet mode does not rely on the disk's response to writes or flush commands to ensure durability of written data; instead it uses CCEs to coerce the disk to order write requests as required for journaling transaction semantics.

B. Design

To explain our design of discreet mode for both ordered and data journaling in ext3, we begin by describing the standard operation of ordered journaling mode and what can go wrong if the disk misbehaves.

Figure 3 shows the required ordering of block updates in a transaction. The updates must be performed in the following order:

1) Data blocks (**D**) are written to their in-place locations on disk
2) Metadata blocks (**M**), such as inodes and bitmaps, are written to the journal
3) Journal commit block (**C**) is written to the journal
4) Metadata blocks are checkpointed and written to their in-place locations on disk
5) Entries from the journal are freed

If a misbehaving disk violates this ordering due to overly aggressive caching, then a crash could result in a variety of problems for the file system and users. For example, if the misbehaving disk first destages the journal blocks to the media and a crash occurs before all of the data blocks have been durably updated (*i.e.*, step 2 before step 1), then recovery will checkpoint metadata blocks to point to old data; thus, a newly updated file may contain corrupted or stale contents.

To ensure that the writes to disk are durably performed even for a misbehaving disk, discreet mode

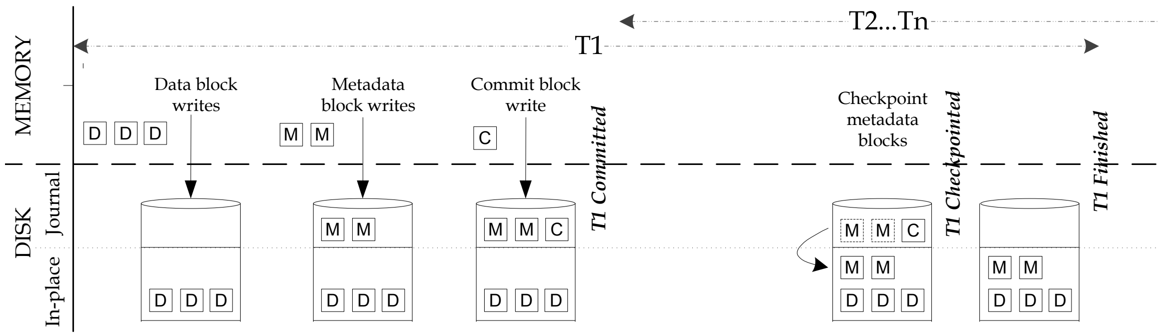

Figure 3. **Block Ordering in ext3 Ordered Journaling Mode .** *The figure shows the ordering of blocks in ext3 ordered journaling mode. Transactions are shown as T1, T2, etc. 'D' denotes data blocks, 'M' metablocks, and 'C' journal commit blocks.*

journaling for ordered mode adds CCE operations after each of the steps one through four. Figure 4 shows the location of CCE operations (or flushzone writes, described below). We emphasize that the addition of CCEs does not affect transaction semantics; CCEs simply ensure that the semantics are maintained even in the presence of misbehaving hardware.

We have also implemented discreet mode for data journaling mode. Data journaling is similar to ordered journaling, except step one listed above is omitted and both data and metadata blocks are written to the journal and checkpointed. For discreet data journaling, CCEs are added after each of the three remaining steps.

C. Implementation

We have implemented discreet mode journaling in both Linux 2.6.13 and 2.6.23. We modified the journaling block device (JBD) layer to include CCE operations at the necessary points described above.

Given a disk, we first analyze it in order to find the most effective flush to be used in the CCE operation. As discussed in Section III, we first fingerprint the disk. We then look for the flush workload that gives the highest probability of eviction at lowest cost. This flush workload is then used in the CCE operation.

To prepare ext3 to operate in discreet mode, we must first allocate space on the disk to which we will direct the flush workload writes. Therefore we have modified the `mke2fs` utility to allocate blocks for the *flushzone*, a region of disk space for the flush traffic. The location of the flush zone has a significant impact on performance; since the flush zone is written at nearly the same time as journal writes, `mke2fs` allocates space for it right after the journal. Like the journal, the flushzone is allocated as a file; a flushzone superblock contains the CCE parameters for the particular disk.

We added functionality in `mount` which reads the on-disk flushzone superblock and stores the flushzone parameters in memory. It notifies the journaling block device (JBD) to operate in discreet mode and also passes the flushzone parameters to it.

V. Evaluation results

We now present performance results for discreet mode journaling implemented in ext3. Specifically, we answer the following questions:

- Is the overhead of discreet mode journaling acceptable? Are there workloads for which it is not?
- How much does an efficient implementation of CCE impact workload performance?

A. Methodology

All our experiments were performed on a machine with a 2.2 GHz Opteron processor, 1 GB RAM, and two 80 GB Hitachi HDS7280SASUN80G disks containing 8 MB of cache. The Hitachi disks were chosen for this evaluation because they happened to be the first disks we acquired; they also represent a favorable case for CCE given their cache-flush cost.

To allow us to generalize our results slightly more to disks with more expensive implementations of CCE, we evaluate two implementations of CCE for the Hitachi disks. The first version issues 128 pseudo-random writes, each of size 12 KB, for a total of 1.5 MB of data. This workload is designed to flush each of the segments of the cache. These random writes flush the cache, but take approximately 170 ms to do so. To show the benefits of optimizing the performance of the CCE, the second version issues the recommended single sequential write of 2.3 MB (see Table IV); this version takes approximately 50 ms.

B. Basic Performance of Discreet Mode

We begin by evaluating the unoptimized versions of CCE within discreet mode journaling. For a range of workloads, we measure three variants of both ordered journaling and data journaling: discreet mode with the disk cache enabled, the regular mode with the disk cache disabled, and the regular (unsafe) mode with disk cache enabled. In the modes where the disk cache is enabled, write-back caching is used.

Since we are targeting the scenario where the caching behavior of a disk is not entirely trusted, we first

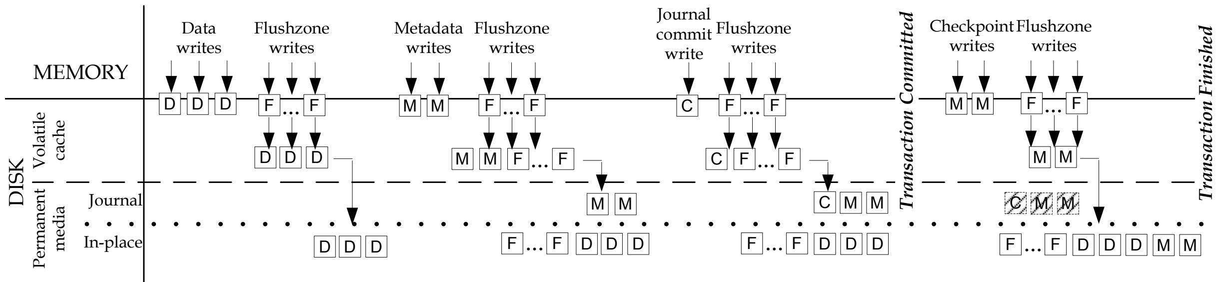

Figure 4. **Ordered journal ext3 in discreet mode.** *The figure shows ext3 in ordered journal mode operating in discreet mode. The extra writes issued to the flushzone are used to flush the blocks from the drive cache to the disk media. The crossed-out journal blocks mean that they are deleted when the blocks are checkpointed to disk.*

Journal type	Journaling mode and cache configuration		
	Regular w/o cache	Discreet (costly flush)	Unsafe w/ cache
Ordered journal	41.97	42.07	41.47
Data journal	42.02	41.64	41.60

Table V : **OpenSSH benchmark runtimes.** *The table shows the runtimes of the OpenSSH benchmark on a HITACHI SATA drive with 8 MB cache. These experiments use the costly CCE and Linux 2.6.13. Run times are shown for both ordered and data journal modes and for regular and discreet mode journaling. For the OpenSSH benchmark, we measure the time to copy, untar, configure, and make the OpenSSH 4.51 source.*

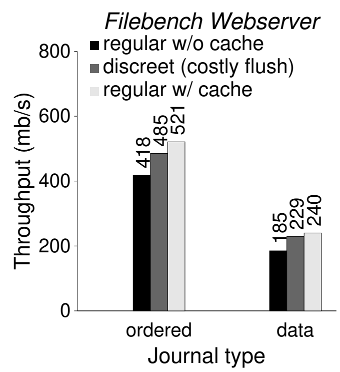

Figure 5. **Performance measurement with Filebench webserver benchmark.** *The graph shows the average throughput in megabits per second for operations in the filebench webserver benchmark on the HITACHI drive (8 MB cache). These experiments use the costly CCE and Linux 2.6.13. The parameters used are: 16 threads, 500 files, and 16KB mean I/O size.*

compare discreet mode to the regular mode of journaling (ordered or data) with the drive's cache disabled. Disabling the drive's cache is currently the only method available for safely updating the file system if one does not completely trust the disk. We note trying to disable the cache of a misbehaving disk is not the same thing as actually disabling it: a misbehaving disk may choose to disregard this command. Therefore, discreet mode may still be the only way to ensure a misbehaving disk performs writes in the required order.

The second comparison is with the unsafe regular mode of journaling (ordered or data) with the drive's cache enabled. This represents the best possible performance for discreet mode journaling and could be achieved if the disk were completely trusted. The difference between the unsafe regular mode with the disk cache enabled and discreet mode is the overhead of not trusting the disk.

OpenSSH: We begin our comparison with the OpenSSH benchmark, which performs a copy, untar, configure, and make of the OpenSSH source code. Table V shows that the discreet mode of both ordered and data journaling performs better than the safe base case of disabling the cache. For this simple workload, discreet mode performs as well as the unsafe regular mode with caching.

Filebench Webserver: We continue with a more I/O-intensive synthetic benchmark that emulates the behavior of a webserver; these results are shown in Figure 5. As expected, all versions of ordered journaling achieve significantly higher throughput than data journaling; this is expected because data journaling writes each data block twice: once to the journal and once to its fixed in-place location on disk.

Within each group, discreet mode again obtains noticeably better throughput than disabling the misbehaving cache; running without the cache degrades the throughput of the application by 15 to 20 percent compared to discreet mode. For this workload, discreet mode performs comparably to the unsafe regular cases, causing a modest 4 to 7 percent drop in throughput.

Postmark benchmark: Figure 6 shows the results of running the Postmark benchmark. Again, as expected, ordered journaling performs better than data journaling across all variants. With ordered mode, discreet mode performs similarly to disabling the cache and about 25% slower than the regular case that trusts the disk. With data journaling, discreet mode consistently performs better than disabling the disk cache; disabling the disk cache incurs approximately a 10% overhead relative to discreet mode.

Completely disabling the disk cache for safety reasons is a poor idea if performance is needed. Even though discreet mode must periodically flush the disk

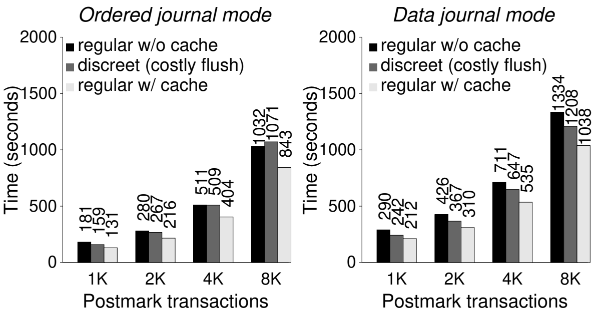

Figure 6. **Performance measurement with Postmark benchmark.** *The graph shows the average times in seconds taken to run the postmark benchmark with varying transactions on a HITACHI drive (8 MB cache). These experiments use the costly CCE and Linux 2.6.23. Run times are shown for regular and discreet mode journaling for both ordered and data journal modes. The parameters: 1000 files (sizes 4KB to 4MB), 1000 to 8000 transactions, and 50/50 read/append and create/delete biases.*

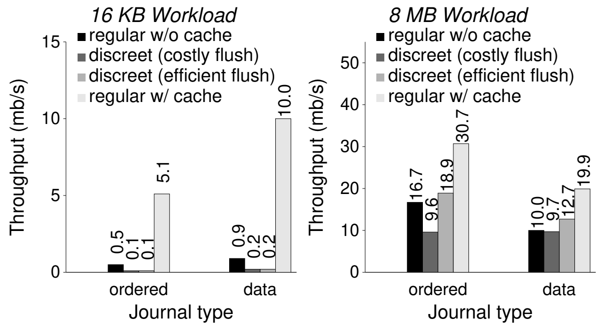

Figure 7. **Filebench varmail: 16 KB vs 8 MB.** *The graphs show the average throughput in megabits per second for operations in the filebench varmail benchmark on the HITACHI drive with Linux 2.6.23. The workload in the first graph appends only 16 KB of data before calling fsync, whereas the workload in the second graph appends 8 MB. The bars illustrate different journaling modes and implementations of the discreet mode with costly (170 ms) or efficient (50 ms) flushes. The benchmark is run with 1 thread and 500 files of 16 KB each.*

cache, it is still able to leverage the cache for write operations that do not require ordering (*e.g.*, updating multiple data blocks). We note that all of the performance results in this section were obtained with the costly version of the CCE taking 170 ms. We investigate the effect of improving the CCE time in our next workloads.

C. Optimizing the Cache Flush

We now investigate a workload that particularly stresses discreet mode journaling: Filebench Varmail. This benchmark repeatedly performs an append operation to a randomly selected file and then issues an `fsync` to ensure that the data is durably updated on disk before proceeding.

This benchmark represents a difficult case for discreet mode journaling because every call to `fsync` causes the file system to flush the disk cache, leading to four complete CCEs as shown in Figure 4.

To improve performance we remove one of the four CCE operations by implementing a *transactional checksum* [24] for discreet mode journaling. A second improvement we implement is to use the optimized CCE operation for the Hitachi disk requiring only 50 ms instead of 170 ms. We implemented these optimizations to offset the effects of frequent `fsync`s. In the general case, discreet mode journaling performs quite well in the absence of these improvements.

The first graph of Figure 7 shows the throughput achieved for the eight journaling variants. As expected for this `fsync` intensive workload, discreet mode journaling performs very poorly and has lower throughput than simply disabling the cache.

To improve the situation, we focus on the latency of the `fsync` operations in the workload; the latency of all other operations are comparable across journaling modes. As stated above, the varmail benchmark calls `fsync` after every append operation. Applications that call `fsync` after small amounts of data perform poorly, regardless of the type of journaling that is used. If one wishes to run `fsync` intensive applications with reasonable performance, a pragmatic approach is to slightly alter the application to group more operations (or larger writes) between calls to `fsync` [15]. We have modified varmail so that instead of appending only 16 KB of data before calling `fsync`, it appends up to 8 MB of data in a larger transaction group before invoking `fsync`.

Figure 8 details the results for the eight journaling variants as the size of the transaction group is increased. As expected, the time for `fsync` grows with the size of the transaction for all journaling modes; however, the time grows more slowly for discreet mode than the others. For regular journaling modes, an `fsync` translates to disk traffic equal to the amount of data just appended; for discreet journaling, an `fsync` translates to disk traffic equal to the appended data plus the flush operations. Thus, for discreet mode, performing larger transactions within the application amortizes the cost of flushing the cache over a larger amount of data.

The second graph of Figure 7 shows the benefit of this simple modification. When the amount of data within each transaction is increased to 8 MB, the performance of all journaling modes improves dramatically (note the difference in the scales across the two graphs). Interestingly, the relative performance of the discreet modes improves the most: with 8 MB application transactions, discreet mode journaling obtains better throughput than disabling the cache for safety. However, it is important to optimize the CCE operation to flush the disk cache efficiently; for example, in ordered mode, the efficient 50 ms flush achieves nearly twice the throughput of the costly 170 ms flush.

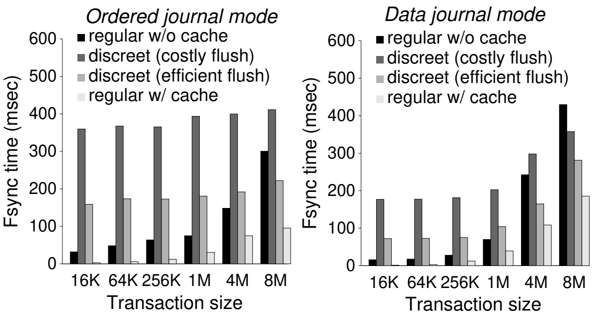

Figure 8. **Filebench varmail: Varying Transaction Size.** *The graph shows the average time in ms for the fsync operation within the varmail benchmark. The amount of data within each transaction is increased from 16 KB to 8 MB. Fsync times are shown for discreet mode journaling with both the efficient and the costly flush (CCE) and for both ordered and data journal modes.*

In conclusion, the performance overhead of discreet mode journaling is highly dependent on the workload. For most I/O workloads (*e.g.*, OpenSSH, the Filebench webserver, and Postmark), the performance of discreet mode journaling (with an unoptimized CCE) is better than the only other (somewhat) safe alternative of disabling the cache.

Small synchronous I/Os cause particularly poor performance for discreet mode; however, if the size of transactions within the application is increased, then discreet mode journaling performs adequately.

We found that it is essential to use a version of CCE that has been optimized for the specific disk cache; using a costly flush requiring 170 ms instead of an optimized flush requiring 50 ms can degrade performance by up to a factor of two. We believe these performance results are favorable enough for us to recommend discreet mode journaling be used when disk trustworthiness is suspect.

VI. Conclusion

File systems are entrusted with data and are expected to provide reliability even when there is unreliable hardware underneath. Unfortunately, disk drives that misbehave with respect to flush commands can prevent file systems from maintaining the expected ordering of durable writes. A disk that misbehaves in order to improve its performance can cause a trusting file system to lose or corrupt data and meta-data.

The only recourse that file systems currently have for coping with misbehaving disks is to disable the disk cache; however, even this command may be ignored by a misbehaving disk. Therefore, we have introduced Coerced Cache Eviction (CCE) to "force" even a misbehaving disk to flush writes to the permanent media. By using the CCE primitive, file systems can ensure that writes are durably updated in the expected order.

As we increasingly adopt the world of "cloud computing" [4], where users rent computing resources and run on virtualized platforms, one needs to attribute more importance to the notion of trust. The host has incentives to efficiently utilize its infrastructure by batching writes across users. Can users trust that their data is flushed to durable media when they are writing to such a virtual medium? We believe that this kind of misbehavior may become more commonplace rather than less, making techniques like CCE more essential.

Acknowledgments

We thank the anonymous reviewers for their feedback and comments, which have substantially improved the content and presentation of this paper. We also thank S. Subramanian, S. Sundaraman, L. Arulraj, M. Saxena, and S. Panneerselvam for their comments on earlier drafts of the paper.

This material is based upon work supported by the National Science Foundation under the following grants: CCF-0811657, CNS-0834392, CCF-0937959, CSR-1017518, as well as by generous donations from NetApp and Google.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or other institutions.

References

[1] D. Anderson, J. Dykes, and E. Riedel. More Than an Interface: SCSI vs. ATA. In *FAST '03*, San Francisco, CA, April 2003.

[2] Anonymous @ Microsoft. Some SATA disks do not allow the file system to force writes to disk properly. Personal Communication.

[3] Anonymous @ Seagate. Some SATA disks (though none from Seagate) do not allow the file system to force writes to disk properly. Personal Communication.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the Clouds: A Berkeley View of Cloud Computing. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[5] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *ISCA '95*, pages 320–331, Santa Margherita Ligure, Italy, June 1995.

[6] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *SIGMETRICS '07*, San Diego, CA, June 2007.

[7] L. N. Bairavasundaram, G. R. Goodson, B. Schroeder, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. An Analysis of Data Corruption in the Storage Stack. In *FAST '08*, pages 223–238, San Jose, California, February 2008.

[8] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://opensolaris.org/os/community/zfs/docs/zfs_last.pdf, 2007.

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

[10] C. Christensen. *The Innovator's Dilemma*. Harper Collins, 2003.

[11] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for Double Disk Failure Correction. In *FAST '04*, pages 1–14, San Francisco, CA, April 2004.

[12] C. Frost, M. Mammarella, E. Kohler, A. de los Reyes, S. Hovsepian, A. Matsuoka, and L. Zhang. Generalized File System Dependencies. In *SOSP '07*, Stevenson, WA, October 2007.

[13] G. R. Ganger and Y. N. Patt. Metadata Update Performance in File Systems. In *OSDI '94*, pages 49–60, Monterey, CA, November 1994.

[14] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *SOSP '03*, pages 29–43, Bolton Landing, NY, October 2003.

[15] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

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

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd edition*. Morgan-Kaufmann, 2002.

[18] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *USENIX Winter '94*, San Francisco, CA, January 1994.

[19] G. F. Hughes and J. F. Murray. Reliability and Security of RAID Storage Systems and D2D Archives Using SATA Disk Drives. *ACM Transactions on Storage*, 1(1):95–107, February 2005.

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

[21] R. Karedla, J. S. Love, and B. G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, 1994.

[22] D. Kotz, S. B. Toh, and S. Radhakrishnan. A Detailed Simulation Model of the HP 97560 Disk Drive. Technical Report TR94-220, Dartmouth College, 1994.

[23] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *FAST '08*, pages 127–141, San Jose, California, February 2008.

[24] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[25] Public forum. IDE write cache and journaling file systems. http://oss.sgi.com/projects/xfs/mail_archive/200204/msg00220.html, Apr. 2002.

[26] Public forum. SATA and Data Corruption. http://www.hardwareanalysis.com/content/topic/25671/, Apr. 2004.

[27] Public forum. Cannot disable write cache on hard drives. http://www.tomshardware.com/forum/227363-46-cannot-disable-write-cache-hard-drives, Jan. 2005.

[28] 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.

[29] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, 1994.

[30] R. H. Saavedra and A. J. Smith. Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes. *IEEE Transactions on Computers*, 44(10):1223–1235, 1995.

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

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

[33] M. Seltzer, P. Chen, and J. Ousterhout. Disk Scheduling Revisited. In *USENIX Winter '90*, pages 313–324, Washington, D.C, January 1990.

[34] E. A. M. Shriver, A. Merchant, and J. Wilkes. An Analytic Behavior Model for Disk Drives with Readahead Caches and Request Reordering. In *SIGMETRICS '98*, Madison, WI, June 1998.

[35] R. Sundaram. The Private Lives of Disk Drives. http://www.netapp.com/go/techontap/matl/sample/0206tot_resiliency.html, February 2006.

[36] 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.

[37] The Serial ATA International Organization. Serial ATA Revision 3.0 Specification. http://www.sata-io.org/technology/6Gbdetails.asp, June 2009.

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

[39] M. Wei, L. M. Grupp, F. E. Spada, and S. Swanson. Reliably erasing data from flash-based solid state drives. In *FAST '11*, San Jose, California, February 2011.

[40] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. In *SIGMETRICS '95*, pages 146–156, Ottawa, Canada, May 1995.