Optimistic Crash Consistency

Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

Department of Computer Sciences
University of Wisconsin, Madison

{vijayc, madthanu, dusseau, remzi}@cs.wisc.edu

Abstract

We introduce *optimistic crash consistency*, a new approach to crash consistency in journaling file systems. Using an array of novel techniques, we demonstrate how to build an optimistic commit protocol that correctly recovers from crashes and delivers high performance. We implement this optimistic approach within a Linux ext4 variant which we call *OptFS*. We introduce two new file-system primitives, `osync()` and `dsync()`, that decouple ordering of writes from their durability. We show through experiments that OptFS improves performance for many workloads, sometimes by an order of magnitude; we confirm its correctness through a series of robustness tests, showing it recovers to a consistent state after crashes. Finally, we show that `osync()` and `dsync()` are useful in atomic file system and database update scenarios, both improving performance and meeting application-level consistency demands.

1 Introduction

Modern storage devices present a seemingly innocuous interface to clients. To read a block, one simply issues a low-level read command and specifies the address of the block (or set of blocks) to read; when the disk finishes the read, it is transferred into memory and any awaiting clients notified of the completion. A similar process is followed for writes.

Unfortunately, the introduction of *write buffering* [28] in modern disks greatly complicates this apparently simple process. With write buffering enabled, disk writes may complete out of order, as a smart disk scheduler may reorder requests for performance [13, 24, 38]; further, the notification received after a write issue implies only that the disk has received the request, not that the data has been written to the disk surface persistently.

Out-of-order write completion, in particular, greatly complicates known techniques for recovering from system crashes. For example, modern journaling file systems such as Linux ext3, XFS, and NTFS all carefully orchestrate a sequence of updates to ensure that writes to main file-system structures and the journal reach disk in a particular order [22]; copy-on-write file systems such as LFS, btrfs, and ZFS also require ordering when updating certain structures. Without ordering, most file systems cannot ensure that state can be recovered after a crash [6].

Write ordering is achieved in modern drives via expensive *cache flush* operations [30]; such flushes cause all buffered dirty data in the drive to be written to the surface (*i.e.*, *persisted*) immediately. To ensure *A* is written before *B*, a client issues the write to *A*, and then a cache flush; when the flush returns, the client can safely assume that *A* reached the disk; the write to *B* can then be safely issued, knowing it will be persisted after *A*.

Unfortunately, cache flushing is expensive, sometimes prohibitively so. Flushes make I/O scheduling less efficient, as the disk has fewer requests to choose from. A flush also unnecessarily forces *all* previous writes to disk, whereas the requirements of the client may be less stringent. In addition, during a large cache flush, disk reads may exhibit extremely long latencies as they wait for pending writes to complete [26]. Finally, flushing conflates ordering and durability; if a client simply wishes to order one write before another, forcing the first write to disk is an expensive manner in which to achieve such an end. In short, the classic approach of flushing is *pessimistic*; it assumes a crash will occur and goes to great lengths to ensure that the disk is never in an inconsistent state via flush commands. The poor performance that results from pessimism has led some systems to disable flushing, apparently sacrificing correctness for performance; for example, the Linux ext3 default configuration did not flush caches for many years [8].

Disabling flushes does not necessarily lead to file system inconsistency, but rather introduces it as a possibility. We refer to such an approach as *probabilistic crash consistency*, in which a crash *might* lead to file system inconsistency, depending on many factors, including workload, system and disk parameters, and the exact timing of

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522726

the crash or power loss. In this paper, one of our first contributions is the careful study of probabilistic crash consistency, wherein we show which exact factors affect the odds that a crash will leave the file system inconsistent (§3). We show that for some workloads, the probabilistic approach rarely leaves the file system inconsistent.

Unfortunately, a probabilistic approach is insufficient for many applications, where certainty in crash recovery is desired. Indeed, we also show, for some workloads, that the chances of inconsistency are high; to realize higher-level application-level consistency (*i.e.*, something a DBMS might desire), the file system must provide something more than probability and chance. Thus, in this paper, we introduce *optimistic crash consistency*, a new approach to building a crash-consistent journaling file system (§4). This optimistic approach takes advantage of the fact that in many cases, ordering can be achieved through other means and that crashes are rare events (similar to optimistic concurrency control [12, 16]). However, realizing consistency in an optimistic fashion is not without challenge; we thus develop a range of novel techniques, including a new extension of the transactional checksum [23] to detect data/metadata inconsistency, delayed reuse of blocks to avoid incorrect dangling pointers, and a selective data journaling technique to handle block overwrite correctly. The combination of these techniques leads to *both* high performance and *deterministic* consistency; in the rare event that a crash does occur, optimistic crash consistency either avoids inconsistency by design or ensures that enough information is present on the disk to detect and discard improper updates during recovery.

We demonstrate the power of optimistic crash consistency through the design, implementation, and analysis of the *optimistic file system (OptFS)*. OptFS builds upon the principles of optimistic crash consistency to implement *optimistic journaling*, which ensures that the file system is kept consistent despite crashes. Optimistic journaling is realized as a set of modifications to the Linux ext4 file system, but also requires a slight change in the disk interface to provide what we refer to as *asynchronous durability notification*, *i.e.*, a notification when a write is persisted in addition to when the write has simply been received by the disk. We describe the details of our implementation (§5) and study its performance (§6), showing that for a range of workloads, OptFS significantly outperforms classic Linux ext4 with pessimistic journaling, and showing that OptFS performs almost identically to Linux ext4 with probabilistic journaling while ensuring crash consistency.

Central to the performance benefits of OptFS is the separation of ordering and durability. By allowing applications to order writes without incurring a disk flush, and request durability when needed, OptFS enables application-level consistency at high performance. OptFS introduces two new file-system primitives: `osync()`, which ensures ordering between writes but only *eventual* durability, and `dsync()`, which ensures immediate durability as well as ordering.

We show how these primitives provide a useful base on which to build higher-level application consistency semantics (§7). Specifically, we show how a document editing application can use `osync()` to implement the atomic update of a file (via a create and then atomic rename), and how the SQLite database management system can use file-system provided ordering to implement ordered transactions with eventual durability. We show that these primitives are sufficient for realizing useful application-level consistency at high performance.

Of course, the optimistic approach, while useful in many scenarios, is not a panacea. If an application requires *immediate, synchronous* durability (instead of eventual, asynchronous durability with consistent ordering), an expensive cache flush is still required. In this case, applications can use `dsync()` to request durability (as well as ordering). However, by decoupling the durability of writes from their ordering, OptFS provides a useful middle ground, thus realizing high performance and meaningful crash consistency for many applications.

2 Pessimistic Crash Consistency

To understand the optimistic approach to journaling, we first describe standard *pessimistic crash consistency* in journaling file systems. To do so, we describe the necessary disk support (*i.e.*, cache-flushing commands) and details on how such crash consistency operates. We then demonstrate the negative performance impact of cache flushing during pessimistic journaling.

2.1 Disk Interface

For the purposes of this discussion, we assume the presence of a disk-level cache flush command. In the ATA family of drives, this is referred to as the "flush cache" command; in SCSI drives, it is known as "synchronize cache". Both operations have similar semantics, forcing all pending dirty writes in the disk to be written to the surface. Note that a flush can be issued as a separate request, or as part of a write to a given block *D*; in the latter case, pending writes are flushed before the write to *D*.

Some finer-grained controls also exist. For example, "force unit access" (FUA) commands read or write around the cache entirely. FUAs are often used in tandem with flush commands; for example, to write *A* before *B*, but to also ensure that both *A* and *B* are durable, a client might write *A*, then write *B* with both cache flushing and FUA enabled; this ensures that when *B* reaches the drive, *A* (and other dirty data) will be forced to disk; subsequently, *B* will be forced to disk due to the FUA.

2.2 Pessimistic Journaling

Given the disk interface described above, we now describe how a journaling file system safely commits data to disk in order to maintain consistency in the event of a system crash. We base our discussion on ordered-mode Linux ext3 and ext4 [34, 35], though much of what we say is applicable to other journaling file systems such as SGI XFS [32], Windows NTFS [27], and IBM JFS [3]. In ordered mode, file-system metadata is journaled to maintain its consistency; data is not journaled, as writing each data block twice reduces performance substantially.

When an application updates file-system state, either metadata, user data, or (often) both need to be updated in a persistent manner. For example, when a user appends a block to a file, a new *data* block (D) must be written to disk (at some point); in addition, various pieces of *metadata* (M) must be updated as well, including the file's inode and a bitmap marking the block as allocated.

We refer to the atomic update of metadata to the journal as a *transaction*. Before committing a transaction Tx to the journal, the file system first writes any data blocks (D) associated with the transaction to their final destinations; writing data before transaction commit ensures that committed metadata does not point to garbage. After these data writes complete, the file system uses the journal to log metadata updates; we refer to these journal writes as J_M. After these writes are persisted, the file system issues a write to a commit block (J_C); when the disk persists J_C, the transaction Tx is said to be *committed*. Finally, after the commit, the file system is free to update the metadata blocks in place (M); if a crash occurs during this *checkpointing* process, the file system can recover simply by scanning the journal and replaying committed transactions. Details can be found elsewhere [22, 35].

We thus have the following set of ordered writes that must take place: D before J_M before J_C before M, or more simply: $D \to J_M \to J_C \to M$. Note that D, J_M, and M can represent more than a single block (in larger transactions), whereas J_C is always a single sector (for the sake of write atomicity). To achieve this ordering, the file system issues a cache flush wherever order is required (*i.e.*, where there is a $\to$ symbol).

Optimizations to this protocol have been suggested in the literature, some of which have been realized in Linux ext4. For example, some have noted that the ordering between data and journaled metadata ($D \to J_M$) is superfluous; removing that ordering can sometimes improve performance ($D|J_M \to J_C \to M$) [22].

Others have suggested a "transactional checksum" [23] which can be used to remove the ordering between the journal metadata and journal commit (J_M and J_C). In the normal case, the file system cannot issue J_M and J_C together, because the drive might reorder them; in that case, J_C might hit the disk first, at which point a system crash (or power loss) would leave that transaction in a seemingly committed state but with garbage contents. By computing a checksum over the entire transaction and placing its value in J_C, the writes to J_M and J_C can be issued together, improving performance; with the checksum present, crash recovery can avoid replay of improperly committed transactions. With this optimization, the ordering is $D \to \overline{J_M|J_C} \to M$ (where the bar over the journal updates indicates their protection via checksum).

Interestingly, these two optimizations do not combine, *i.e.*, $D|\overline{J_M|J_C} \to M$ is not correct; if the file system issues D, J_M, and J_C together, it is possible that J_M and J_C reach the disk first. In this case, the metadata commits before the data; if a crash occurs before the data is written, an inode (or indirect block) in the committed transaction could end up pointing to garbage data. Oddly, ext4 allows this situation with the "right" set of mount options.

We should note that one other important ordering exists among updates, specifically the order *between* transactions; journaling file systems assume transactions are committed to disk in order (*i.e.*, $Tx_i \to Tx_{i+1}$) [35]. Not following this ordering could lead to odd results during crash recovery. For example, a block B could have been freed in Tx_i, and then reused in Tx_{i+1}; in this case, a crash after Tx_{i+1} committed but before Tx_i did would lead to a state where B is allocated to two files.

Finally, and most importantly, we draw attention to the *pessimistic* nature of this approach. Whenever ordering is required, an expensive cache flush is issued, thus forcing *all* pending writes to disk, when perhaps only a subset of them needed to be flushed. In addition, the flushes are issued even though the writes may have gone to disk in the correct order anyhow, depending on scheduling, workload, and other details; flushes induce extra work that may not be necessary. Finally, and perhaps most harmful, is the fact that the burden of flushes is added despite the fact that crashes are rare, thus exacting a heavy cost in anticipation of an extremely occasional event.

2.3 Flushing Performance Impact

To better understand the performance impact of cache flushing during pessimistic journaling, we now perform a simple experiment. Specifically, we run the Varmail benchmark atop Linux ext4, both with and without cache flushing; with cache flushing enabled, we also enable transactional checksums to see their performance impact. Varmail is a good choice here as it simulates an email server and includes many small synchronous updates to disk, thus stressing the journaling machinery described above. The experimental setup for this benchmark and configuration is described in more detail in Section 6.

From Figure 1, we can observe the following. First, transactional checksums increase performance slightly,

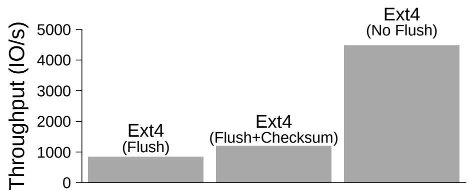

Figure 1: **The Cost of Flushing.** *The figure shows the performance of Filebench Varmail on different ext4 configurations. Performance increases 5X when flushes are disabled.*

showing how removing a single ordering point (via a checksum) can help. Second, and most importantly, there is a vast performance improvement when cache flushing is turned off, in this case nearly a factor of five. Given this large performance difference, in some installations, cache flushing is disabled, which leads to the following question: what kind of crash consistency is provided when flushing is disabled? Surprisingly, the answer is not “none”, as we now describe.

3 Probabilistic Crash Consistency

Given the potential performance gains, practitioners sometimes forgo the safety provided by correct implementations that issue flushes and choose to disable flushes [8]. In this fast mode, a risk of file-system inconsistency is introduced; if a crash occurs at an untimely point in the update sequence, and blocks have been reordered across ordering points, crash recovery as run by the file system will result in an inconsistent file system.

In some cases, practitioners observed that skipping flush commands sometimes did *not* lead to observable inconsistency, despite the presence of (occasional) crashes. Such commentary led to a debate within the Linux community as to underlying causes. Long-time kernel developer Theodore Ts'o hypothesized why such consistency was often achieved despite the lack of ordering enforcement by the file system [33]:

> I suspect the real reason why we get away with it so much with ext3 is that the journal is usually contiguous on disk, hence, when you write to the journal, it's highly unlikely that commit block will be written and the blocks before the commit block have not. ... The most important reason, though, is that the blocks which are dirty don't get flushed out to disk right away!

What the Ts'o Hypothesis refers to specifically is two orderings: $J_M \rightarrow J_C$ and $J_C \rightarrow M$. In the first case, Ts'o notes that the disk is likely to commit J_C to disk after J_M even without an intervening flush (note that this is without the presence of transactional checksums) due to *layout* and *scheduling*; disks are simply unlikely to reorder two writes that are contiguous. In the second case, Ts'o notes that $J_C \rightarrow M$ often holds without a flush due to *time*; the checkpoint traffic that commits M to disk often occurs long after the transaction has been committed, and thus ordering is preserved without a flush.

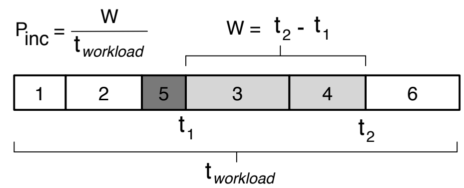

Figure 2: **The Probability of Inconsistency (P_{inc}).** *An example of a window of vulnerability is shown. Blocks 1 through 6 were meant to be written in strict order to disk. However, block 5 (dark gray) is written early. Once 5 is committed, a window of vulnerability exists until blocks 3 and 4 (light gray) are committed; a crash during this time will lead to observable reordering. The probability of inconsistency is calculated by dividing the time spent in such a window (i.e., $W = t_2 - t_1$) by the total runtime of the workload (i.e., $t_{workload}$).*

We refer to this arrangement as *probabilistic consistency*. In such a configuration, typical operation may or may not result in much reordering, and thus the disk is only sometimes in an inconsistent state. A crash may not lead to inconsistency despite a lack of enforcement by the file system via flush commands. Despite probabilistic crash consistency offering *no* guarantees on consistency after a crash, many practitioners are drawn to it due to large performance gains from turning off flushing.

3.1 Quantifying Probabilistic Consistency

Unfortunately, probabilistic consistency is not well understood. To shed light on this issue, we quantify how often inconsistency arises without flushing via simulation. To do so, we disable flushing in Linux ext4 and extract block-level traces underneath ext4 across a range of workloads. We analyze the traces carefully to determine the chances of an inconsistency occurring due to a crash. Our analysis is done via a simulator built atop DiskSim [4], which enables us to model complex disk behaviors (*e.g.*, scheduling, caching).

The main output of our simulations is a determination of when a *window of vulnerability* (W) arises, and for how long such windows last. Such a window occurs due to reordering. For example, if A should be written to disk before B, but B is written at time t_1 and A written at t_2, the state of the system is vulnerable to inconsistency in the time period between, $W = t_2 - t_1$. If the system crashes during this window, the file system will be left in an inconsistent state; conversely, once the latter block (A) is written, there is no longer any concern.

Given a workload and a disk model, it is thus possible to quantify the *probability of inconsistency (P_{inc})* by dividing the total time spent in windows of vulnerability by the total run time of the workload ($P_{inc} = \cup W_i / t_{workload}$); Figure 2 shows an example. Note that when a workload is run on a file system with cache-flushing enabled, P_{inc} is always zero.

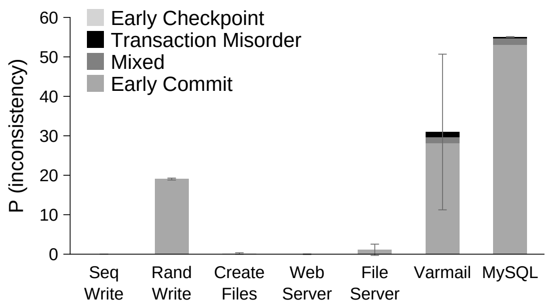

Figure 3: **Workload Study of P_{inc}.** *The figure shows P_{inc} for six workloads. The first two workloads are sequential and random writes to a 1 GB file. Fileserver, Webserver, and Varmail are part of the Filebench benchmark suite [17]. Fileserver performs a sequence of creates, deletes, appends, reads, and writes. Webserver emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append. Varmail emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. MySQL represents the OLTP benchmark from Sysbench [1]. Each bar is broken down into the percent contribution of the different types of misordering. Standard deviations are shown as well.*

3.2 Factors affecting P_{inc}

We now explore P_{inc} more systematically. Specifically, we determine sensitivity to workload and disk parameters such as queue size and placement of the journal relative to file-system structures. We use the validated Seagate Cheetah 15k.5 disk model [4] provided with DiskSim for our experiments.

3.2.1 Workload

We first show how the workload can impact P_{inc}. For this experiment, we use 6 different workloads described in the caption of Figure 3. From the figure, we make the following observations. Most importantly, P_{inc} is workload dependent. For example, if a workload is mostly read oriented, there is little chance of inconsistency, as file-system state is not updated frequently (*e.g.*, Webserver). Second, for write-heavy workloads, the *nature* of the writes is important; workloads that write randomly or force writes to disk via `fsync()` lead to a fairly high chance of a crash leaving the file system inconsistent (*e.g.*, random writes, MySQL, Varmail). Third, there can be high variance in P_{inc}; small events that change the order of persistence of writes can lead to large differences in chances of inconsistency. Finally, even under extreme circumstances, P_{inc} never reaches 100% (the graph is cut off at 60%); there are many points in the lifetime of a workload when a crash will not lead to inconsistency.

Beyond the overall P_{inc} shown in the graph, we also break the probability further by the *type* of reordering that leads to a window of vulnerability. Specifically, assuming the following commit ordering ($D|J_M \rightarrow J_C \rightarrow M$), we determine when a particular reordering (*e.g.*, J_C before D) has resulted. The graph breaks down P_{inc} into these fine-grained reordering categories, grouped into the following relevant cases: *early commit* (*e.g.*, $J_C \rightarrow J_M|D$), *early checkpoint* (*e.g.*, $M \rightarrow D|J_M|J_C$), *transaction misorder* (*e.g.*, $Tx_i \rightarrow Tx_{i-1}$), and *mixed* (*e.g.*, where more than one category could be attributed).

Our experiments show that early commit before data, ($J_C \rightarrow D$), is the largest contributor to P_{inc}, accounting for over 90% of inconsistency across all workloads, and 100% in some cases (Fileserver, random writes). This is not surprising, as in cases where transactions are being forced to disk (*e.g.*, due to calls to `fsync()`), data writes (D) are issued just before transaction writes (J_M and J_C); slight re-orderings by the disk will result in J_C being persisted first. Also, for some workloads (MySQL, Varmail), *all* categories might contribute; though rare, early checkpoints and transaction misordering can arise. Thus, any approach to provide reliable consistency mechanisms must consider all possible causes, not just one.

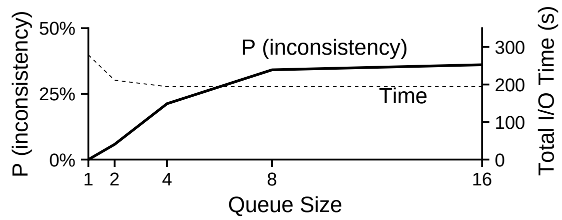

Figure 4: **The Effect of Queue Size.** *The figure shows P_{inc} (left y-axis) and total I/O completion time (right y-axis) as the queue size of the simulated disk varies (x-axis). For this experiment, we use the Varmail workload.*

3.2.2 Queue Size

For the remaining studies, we focus on Varmail, as it exhibits the most interesting and varied probability of inconsistency. First, we show how disk scheduler queue depth matters. Figure 4 plots the results of our experiment. The left y-axis plots P_{inc} as we vary the number of outstanding requests to the disk; the right y-axis plots performance (overall time for all I/Os to complete).

From the figure, we observe the following three results. First, when there is no reordering done by the disk (*i.e.*, queue size is 1), there is no chance of inconsistency, as writes are committed in order; we would find the same result if we used FIFO disk scheduling (instead of SPTF). Second, even with small queues (*e.g.*, 8), a great deal of inconsistency can arise; one block committed too early to disk can result in very large windows of vulnerability. Finally, we observe that a modest amount of reordering does indeed make a noticeable performance difference; in-disk SPTF scheduling improves performance by about 30% with a queue size of 8 or more.

3.2.3 Journal Layout

We now study how distance between the main file-system structures and the journal affects P_{inc}. Figure 5

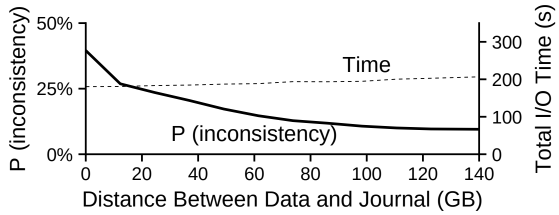

Figure 5: **The Effect of Distance.** *The figure shows P_{inc} (left y-axis) and total I/O completion time (right y-axis) as the distance (in GB) between the data region and the journal of the simulated disk is increased (x-axis). For this experiment, we use the Varmail workload, with queue size set to 8.*

plots the results of varying the location of Varmail's data and metadata structures (which are usually located in one disk area) from close to the journal (left) to far away.

From the figure, we observe distance makes a significant difference in P_{inc}. Recall that one of the major causes of reordering is early commit (*i.e.*, J_C written before D); by separating the location of data and the journal, it becomes increasingly unlikely for such reordering to occur. Secondly, we also observe that increased distance is not a panacea; inconsistency (10%) still arises for Varmail. Finally, increased distance from the journal can affect performance somewhat; there is a 14% decrease in performance when moving Varmail's data and metadata from right next to the journal to 140 GB away.

We also studied a number of other factors that might affect P_{inc}, including the placement of the journal as it relates to track boundaries on the disk, and other potential factors. In general, these parameters did not significantly affect P_{inc} and thus are not included.

3.3 Summary

The classic approach to journaling is overly pessimistic, forcing writes to persistent storage often when only ordering is desired. As a result, users have sometimes turned to probabilistic journaling, taking their chances with consistency in order to gain more performance. We have carefully studied which factors affect the consistency of the probabilistic approach, and shown that for some workloads, it works fairly well; unfortunately, for other workloads with a high number of random-write I/Os, or where the application itself forces traffic to disk, the probability of inconsistency becomes high. As devices become more sophisticated, and can handle a large number of outstanding requests, the odds that a crash will cause inconsistency increases. Thus, to advance beyond the probabilistic approach, a system must include machinery to either avoid situations that lead to inconsistency, or be able to detect and recover when such occurrences arise. We now describe one such approach: *optimistic crash consistency.*

4 Optimistic Crash Consistency

Given that journaling with probabilistic consistency often gives consistent results even in the presence of system crashes, we note a new opportunity. The goal of *optimistic crash consistency*, as realized in an *optimistic journaling* system, is to commit transactions to persistent storage in a manner that maintains consistency to the same extent as pessimistic journaling, but with nearly the same performance as with probabilistic consistency. Optimistic journaling requires minimal changes to current disk interfaces and the journaling layer; in particular, our approach does not require changes to file-system structures outside of the journal (*e.g.*, backpointers [6]).

To describe optimistic crash consistency and journaling, we begin by describing the intuition behind optimistic techniques. Optimistic crash consistency is based on two main ideas. First, checksums can remove the need for ordering writes. Optimistic crash consistency eliminates the need for ordering during transaction commit by generalizing metadata transactional checksums [23] to include data blocks. During recovery, transactions are discarded upon checksum mismatch.

Second, *asynchronous durability notifications* are used to delay checkpointing a transaction until it has been committed durably. Fortunately, this delay does not affect application performance, as applications block until the transaction is committed, not until it is checkpointed. Additional techniques are required for correctness in scenarios such as block reuse and overwrite.

We first propose an additional notification that disk drives should expose. We then explain how optimistic journaling provides different properties to preserve the consistency semantics of ordered journaling. We show that these properties can be achieved using a combination of optimistic techniques. We also describe an additional optimistic technique which enables optimistic journaling to provide consistency equivalent to data journaling.

4.1 Asynchronous Durability Notification

The current interface to the disk for ensuring that write operations are performed in a specified order is pessimistic: the upper-level file system tells the lower-level disk when it must flush its cache (or certain blocks) and the disk must then promptly do so. However, the actual ordering and durability of writes to the platter does not matter, unless there is a crash. Therefore, the current interface is overly constraining and limits I/O performance.

Rather than requiring the disk to obey ordering and durability commands from the layer above, we propose that the disk be freed to perform reads and writes in the order that optimizes its scheduling and performance. Thus, the performance of the disk is optimized for the common case in which there is no crash.

Given that the file system must still be able to guarantee consistency and durability in the event of a crash, we propose a minimal extension to the disk interface. With an *asynchronous durability notification* the disk informs the upper-level client that a specific write request has completed and is now guaranteed to be durable. Thus there will be two notifications from the disk: first when the disk has received the write and later when the write has been persisted. Some interfaces, such as Forced Unit Access (FUA), provide a single, *synchronous* durability notification: the drive receives the request and indicates completion when the request has been persisted [14, 37]. Tagged Queuing allows a limited number of requests to be outstanding at a given point of time [15, 37]. Unfortunately, many drives do not implement tagged queuing and FUA reliably [18]. Furthermore, a request tagged with FUA also implies urgency, prompting some implementations to force the request to disk immediately. While a correct implementation of tagged queuing and FUA may suffice for optimistic crash consistency, we feel that an interface that decouples request acknowledgement from persistence enables higher levels of I/O concurrency and thus provides a better foundation on which to build OptFS.

4.2 Optimistic Consistency Properties

As described in Section 2.2, ordered journaling mode involves the following writes for each transaction: data blocks, D, to in-place locations on disk; metadata blocks to the journal, J_M; a commit block to the journal, J_C; and finally, a checkpoint of the metadata to its in-place location, M. We refer to writes belonging to a particular transaction i with the notation $:i$; for example, the journaled metadata of transaction i is denoted $J_M\!:i$.

Ordered journaling mode ensures several properties. First, metadata written in transaction $Tx\!:i+1$ cannot be observed unless metadata from transaction $Tx\!:i$ is also observed. Second, it is not possible for metadata to point to invalid data. These properties are maintained by the recovery process and how writes are ordered. If a crash occurs after the transaction is properly committed (*i.e.*, D, J_M, and J_C are all durably written), but before M is written, then the recovery process can replay the transaction so that M is written to its in-place location. If a crash occurs before the transaction is completed, then ordered journaling ensures that no in-place metadata related to this transaction was updated.

Optimistic journaling allows the disk to perform writes in any order it chooses, but ensures that in the case of a crash, the necessary consistency properties are upheld for ordered transactions. To give the reader some intuition for why particular properties are sufficient for ordered journaling semantics, we walk through the example in Figure 6. For simplicity, we begin by assuming that data

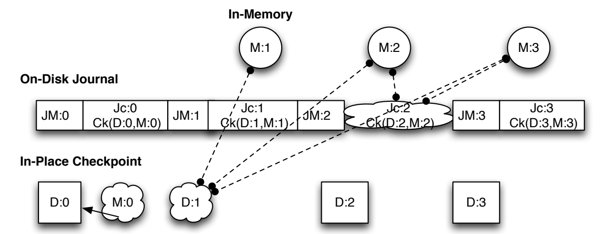

Figure 6: **Optimistic Journaling.** *The figure shows four transactions in progress, involving writes to main memory, the on-disk journal, and to in-place checkpoints on disk. A rectangle block indicates that the file system has been notified that the write has been durably completed. Cloud-shaped blocks indicate that the write has been initiated, but the file system has not yet been notified of its completion and it may or may not be durable. Circles indicate dirty blocks in main memory that cannot be written until a previous write is durable; a dashed line indicates the write it is dependent on. Finally, the solid arrow indicates that the meta-data may refer to the data block.*

blocks are not reused across transactions (*i.e.*, they are not freed and re-allocated to different files or overwritten); we will remove this assumption later (§4.3.5).

In Figure 6, four transactions are in progress: $Tx\!:0$, $Tx\!:1$, and $Tx\!:2$, and $Tx\!:3$. At this point, the file system has received notification that $Tx\!:0$ is durable (*i.e.*, $D\!:0$, $J_M\!:0$, and $J_C\!:0$) and so it is in the process of checkpointing the metadata $M\!:0$ to its in-place location on disk (note that $M\!:0$ may point to data $D\!:0$). If there is a crash at this point, the recovery mechanism will properly replay $Tx\!:0$ and re-initiate the checkpoint of $M\!:0$. Since $Tx\!:0$ is durable, the application that initiated these writes can be notified that the writes have completed (*e.g.*, if it called `fsync()`). Note that the journal entries for $Tx\!:0$ can finally be freed once the file system has been notified that the in-place checkpoint write of $M\!:0$ is durable.

The file system has also started transactions $Tx\!:1$ through $Tx\!:3$; many of the corresponding disk writes have been initiated, while others are being held in memory based on unresolved dependencies. Specifically, the writes for $D\!:1$, $J_M\!:1$, and $J_C\!:1$ have been initiated; however, $D\!:1$ is not yet durable, and therefore the metadata ($M\!:1$), which may refer to it, cannot be checkpointed. If $M\!:1$ were checkpointed at this point and a crash occurred (with $M\!:1$ being persisted and $D\!:1$ not), $M\!:1$ could be left pointing to garbage values for $D\!:1$. If a crash occurs now, before $D\!:1$ is durable, checksums added to the commit block of $Tx\!:1$ will indicate a mismatch with $D\!:1$; the recovery process will not replay $Tx\!:1$, as desired.

$Tx\!:2$ is allowed to proceed in parallel with $Tx\!:1$; in this case, the file system has not yet been notified that the journal commit block $J_C\!:2$ has completed; again, since the transaction is not yet durable, metadata $M\!:2$ cannot be checkpointed. If a crash occurs when $J_C\!:2$ is not yet durable, then the recovery process will detect a mismatch between the data blocks and the checksums and not replay $Tx{:}2$. Note that $D{:}2$ may be durable at this point with no negative consequences because no metadata is allowed to refer to it yet, and thus it is not reachable.

Finally, $Tx{:}3$ is also in progress. Even if the file system is notified that $D{:}3$, $J_M{:}3$, and $J_C{:}3$ are all durable, the checkpoint of $M{:}3$ cannot yet be initiated because essential writes in $Tx{:}1$ and $Tx{:}2$ are not durable (namely, $D{:}1$ and $J_C{:}2$). $Tx{:}3$ cannot be made durable until all previous transactions are guaranteed to be durable; therefore, its metadata $M{:}3$ cannot be checkpointed.

4.3 Optimistic Techniques

The behavior of optimistic journaling described above can be ensured with a set of optimistic techniques: in-order journal recovery and release, checksums, background writes after notification, reuse after notification, and selective data journaling. We now describe each.

4.3.1 In-Order Journal Recovery

The most basic technique for preserving the correct ordering of writes after a crash occurs during the journal recovery process itself. The recovery process reads the journal to observe which transactions were made durable and it simply discards or ignores any write operations that occurred out of the desired ordering.

The correction that optimistic journaling applies is to ensure that if any part of a transaction $Tx{:}i$ was not correctly or completely made durable, then neither transaction $Tx{:}i$ nor any following transaction $Tx{:}j$ where $j > i$ is left durable. Thus, journal recovery must proceed in-order, sequentially scanning the journal and performing checkpoints in-order, stopping at the first transaction that is not complete upon disk. The in-order recovery process will use the checksums described below to determine if a transaction is written correctly and completely.

4.3.2 In-Order Journal Release

Given that completed, durable journal transactions define the write operations that are durable on disk, optimistic journaling must ensure that journal transactions are not freed (or overwritten) until all corresponding checkpoint writes (of metadata) are confirmed as durable.

Thus, optimistic journaling must wait until it has been notified by the disk that the checkpoint writes corresponding to this transaction are durable. At this point, optimistic journaling knows that the transaction need not be replayed if the system crashes; therefore, the transaction can be released. To preserve the property that $Tx{:}i+1$ is made durable only if $Tx{:}i$ is durable, transactions must be freed in order.

4.3.3 Checksums

Checksums are a well-known technique for detecting data corruption and lost writes [20,31]. A checksum can also be used to detect whether or not a write related to a specific transaction has occurred. Specifically, checksums can optimistically "enforce" two orderings: that the journal commit block (J_C) persists only after metadata to the journal (J_M) and after data blocks to their in-place location (D). This technique for ensuring metadata is durably written to the journal in its entirety has been referred to as transactional checksumming [23]; in this approach, a checksum is calculated over J_M and placed in J_C. If a crash occurs during the commit process, the recovery procedure can reliably detect the mismatch between J_M and the checksum in J_C and not replay that transaction (or any transactions following). To identify this particular instance of transactional checksumming we refer to it as *metadata transactional checksumming*.

A similar, but more involved, version of *data transactional checksumming* can be used to ensure that data blocks D are written in their entirety as part of the transaction. Collecting these data blocks and calculating their checksums as they are dirtied in main memory is more involved than performing the checksums over J_M, but the basic idea is the same. With the data checksums and their on-disk block addresses stored in J_C, the journal recovery process can abort transactions upon mismatch. Thus, data transactional checksums enable optimistic journaling to ensure that metadata is not checkpointed if the corresponding data was not durably written.

4.3.4 Background Write after Notification

An important optimistic technique ensures that the checkpoint of the metadata (M) occurs after the preceding writes to the data and the journal (*i.e.*, D, J_M, and J_C). While pessimistic journaling guaranteed this behavior with a flush after J_C, optimistic journaling explicitly postpones the checkpoint write of metadata M until it has been notified that all previous transactions have been durably completed. Note that it is not sufficient for M to occur after only J_C; D and J_M must precede M as well since optimistic journaling must ensure that the entire transaction is valid and can be replayed if any of the in-place metadata M is written. Similarly, $M{:}i+1$ must be postponed until all transactions $Tx{:}i$ have been durably committed to ensure that $M{:}i+1$ is not durable if $M{:}i$ cannot be replayed. We note that $M{:}i+1$ does not need to wait for $M{:}i$ to complete, but must instead wait for the responsible transaction $Tx{:}i$ to be durable.

Checkpointing is one of the few points in the optimistic journaling protocol where the file system must wait to issue a particular write until a specific set of writes have completed. However, this particular waiting is not likely to impact performance because checkpointing occurs in the background. Subsequent writes by applications will be placed in later transactions and these journal updates can be written independently of any other outstanding writes; journal writes do not need to wait for previous checkpoints or transactions. Therefore, even applications waiting for journal writes (*e.g.*, by calling `fsync()`) will not observe the checkpoint latency.

For this reason, waiting for the asynchronous durability notification before a background checkpoint is fundamentally more powerful than the pessimistic approach of sending an ordering command to the disk (*i.e.*, a cache flush). With a traditional ordering command, the disk is not able to postpone checkpoint writes across multiple transactions. On the other hand, the asynchronous durability notification command does not artificially constrain the ordering of writes and gives more flexibility to the disk so that it can best cache and schedule writes; the command also provides the needed information to the file system so that it can allow independent writes to proceed while appropriately holding back dependent writes.

4.3.5 Reuse after Notification

The preceding techniques were sufficient for handling the cases where blocks were not reused across transactions. The difficulty occurs with ordered journaling because data writes are performed to their in-place locations, potentially overwriting data on disk that is still referenced by durable metadata from previous transactions. Therefore, additional optimistic techniques are needed to ensure that durable metadata from earlier transactions never points to incorrect data blocks changed in later transactions. This is a security issue: if user A deletes their file, and then the deleted block becomes part of user B's file, a crash should not lead to user A being able to view user B's data. This is also one of the *update dependency* rules required for Soft Updates [25], but optimistic journaling enforces this rule with a different technique: reuse after notification.

To understand the motivation for this technique, consider the steps when a data block D_A is freed from one file M_A and allocated to another file, M_B and rewritten with the contents D_B. Depending on how writes are reordered to disk, a durable version of M_A may point to the erroneous content of D_B.

This problem can be fixed with transactions as follows. First, the freeing of D_A and update to M_A, denoted $M_{A'}$, is written as part of a transaction $J_{M_{A'}}\!:i$; the allocation of D_B to M_B is written in a later transaction as $D_B\!:i+1$ and $J_{M_B}\!:i+1$. Pessimistic journaling ensures that $J_{M_{A'}}\!:i$ occurs before $D_B\!:i+1$ with a traditional flush between every transaction. The optimistic techniques introduced so far are not sufficient to provide this guarantee because the writes to $D_B\!:i+1$ in their in-place locations cannot be recovered or rolled back if M'_A is lost (even if there is a checksum mismatch and transactions $Tx\!:i$ or $Tx\!:i+1$ are found to be incomplete).

Optimistic journaling guarantees that $J_{M_{A'}}\!:i$ occurs before $D_B\!:i+1$ by ensuring that data block D_A is not reallocated to another file until the file system has been notified by the disk that $J_{M_{A'}}\!:i$ has been durably written; at this point, the data block D_A is "durably free." When the file M_B must be allocated a new data block, the optimistic file system allocates a "durably-free" data block that is known to not be referenced by any other files; finding a durably-free data block is straight-forward given the proposed asynchronous durability notification from disks.

Performing reuse only after notification is unlikely to cause the file system to wait or to harm performance. Unless the file system is nearly 100% full, there should always exist a list of data blocks that are known to be durably free; under normal operating conditions, the file system is unlikely to need to wait to be informed by the disk that a particular data block is available.

4.3.6 Selective Data Journaling

Our final optimistic technique selectively journals data blocks that have been overwritten. This technique allows optimistic journaling to provide data journaling consistency semantics instead of ordered journaling semantics.

A special case of an update dependency occurs when a data block is overwritten in a file and the metadata for that file (*e.g.*, size) must be updated consistently. Optimistic journaling could handle this using reuse after notification: a *new* durably-free data block is allocated to the file and written to disk ($D_B\!:j$), and then the new metadata is journaled ($J_{M_B}\!:j$). The drawback of this approach is that the file system takes on the behavior of a copy-on-write file system and loses some of the locality benefits of an update-in-place file system [21]; since optimistic journaling forces a durably-free data block to be allocated, a file that was originally allocated contiguously and provided high sequential read and write throughput may lose its locality for certain random-update workloads.

If update-in-place is desired for performance, a different technique can be used: *selective* data journaling. Data journaling places both metadata and data in the journal and both are then updated in-place at checkpoint time. The attractive property of data journaling is that in-place data blocks are not overwritten until the transaction is checkpointed; therefore, data blocks can be reused if their metadata is also updated in the same transaction. The disadvantage of data journaling is that every data block is written twice (once in the journal, J_D, and once in its checkpointed in-place location, D) and therefore often has worse performance than ordered journaling [22].

Selective data journaling allows ordered journaling to be used for the common case and data journaling only when data blocks are repeatedly overwritten within the same file and the file needs to maintain its original layout on disk. In selective data journaling, the checkpoint of both D and M simply waits for durable notification of all the journal writes (J_D, J_M, and J_C).

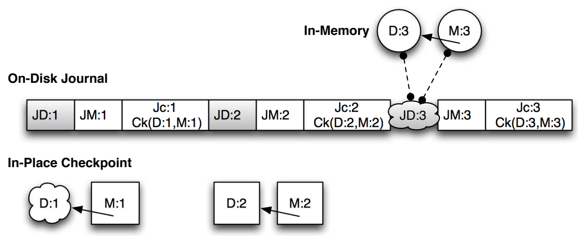

Figure 7: **Optimistic Journaling: Selective Data Journaling.** *The figure shows that selective data journaling may be used when transactions involve overwriting in-place data. Data blocks are now placed in the journal and checkpointed after the transaction is committed.*

Figure 7 shows an example of how selective data journaling can be used to support overwrite, in particular the case where blocks are reused from previous transactions without clearing the original references to those blocks. In this example, data blocks for three files have been overwritten in three separate transactions.

The first transaction illustrates how optimistic ordering ensures that durable metadata does not point to garbage data. After the file system has been notified of the durability of $Tx{:}1$ (specifically, of $J_D{:}1$, $J_M{:}1$, and $J_C{:}1$), it may checkpoint both $D{:}1$ and $M{:}1$ to their in-place locations. Because the file system can write $M{:}1$ without waiting for a durable notification of $D{:}1$, in the case of crash it is possible for $M{:}1$ to refer to garbage values in $D{:}1$; however, the recovery process will identify this situation due to the checksum mismatch and replay $Tx{:}1$ with the correct values for $D{:}1$.

The second and third transactions illustrate how optimistic ordering ensures that later writes are visible only if all earlier writes are visible as well. Specifically, $D{:}2$ and $M{:}2$ have been checkpointed, but only because both $Tx{:}2$ and $Tx{:}1$ are both durable; therefore, a client cannot see new contents for the second file without seeing new contents for the first file. Furthermore, neither $D{:}3$ nor $M{:}3$ (or any later transactions) can be checkpointed yet because not all blocks of its transaction are known to be durable. Thus, selective data journaling provides the desired consistency semantics while allowing overwrites.

4.4 Durability vs. Consistency

Optimistic journaling uses an array of novel techniques to ensure that writes to disk are properly ordered, or that enough information exists on disk to recover from an untimely crash when writes are issued out of order; the result is file-system consistency and proper ordering of writes, but without guarantees of durability. However, some applications may wish to force writes to stable storage for the sake of durability, not ordering. In this case, something more than optimistic ordering is needed; the file system must either wait for such writes to be persisted (via an asynchronous durability notification) or issue flushes to force the issue. To separate these cases, we believe two calls should be provided; an "ordering" sync, `osync()`, guarantees ordering between writes, while a "durability" sync, `dsync()`, ensures when it returns that pending writes have been persisted.

We now define and compare the guarantees given by `osync()` and `dsync()`. Assume the user makes a series of writes $W_1, W_2, ..., W_n$. If no `osync()` or `dsync()` calls are made, there is no guarantee as to file-system state after a crash: any or all of the updates may be lost, and updates may be applied out of order, *i.e.*, W_2 may be applied without W_1.

Now consider when every write is followed by `dsync()`, *i.e.*, $W_1, d_1, W_2, d_2, ..., W_n, d_n$. If a crash happens after d_i, the file system will recover to a state with $W_1, W_2, ..., W_i$ applied.

If every write was followed by `osync()`, *i.e.*, $W_1, o_1, W_2, o_2, ..., W_n, o_n$, and a crash happens after o_i, the file system will recover to a state with $W_1, W_2, ..., W_{i-k}$ applied, where the last k writes had not been made durable before the crash. We term this *eventual* durability. Thus `osync()` provides *prefix semantics* [36], ensuring that users always see a consistent version of the file system, though the data may be stale. Prior research indicates that this is useful in many domains [7].

5 Implementation of OptFS

We have implemented the Optimistic File System (OptFS) inside Linux 3.2, based on the principles outlined before (§4), as a variant of the ext4 file system, with additional changes to the JBD2 journaling layer and virtual memory subsystem.

5.1 Asynchronous Durability Notifications

Since current disks do not implement the proposed asynchronous durability notification interface, OptFS uses an approximation: *durability timeouts*. Durability timeouts represent the maximum time interval that the disk can delay committing a write request to the non-volatile platter. When a write for block A is received at time T by the disk, the block must be made durable by time $T + T_D$. The value of T_D is specific to each disk, and must be exported by the disk to higher levels in the storage stack.

Upon expiration of the time interval T_D, OptFS considers the block to be durable; this is equivalent to the disk notifying OptFS after T_D seconds. Note that to account for other delays in the I/O subsystem, T_D is measured from the time the write is acknowledged by the disk, and not from the time the write is issued by the file system.

This approximation is limiting; T_D might overestimate the durability interval, leading to performance problems and memory pressure; T_D might underestimate the durability interval, comprising consistency. OptFS errs towards safety and sets T_D to be 30 seconds.

5.2 Handling Data Blocks

OptFS does not journal *all* data blocks: newly allocated data blocks are only checksummed; their contents are not stored in the journal. This complicates journal recovery as data-block contents may change after the checksum was recorded. Due to selective data journaling, a data block that is not journaled (as it is newly allocated) in one transaction might be overwritten in the following transaction and therefore journaled. For example, consider data block D with content A belonging to Tx_1. The checksum A will be recorded in Tx_1. D is overwritten by Tx_2, with content B. Though this sequence is valid, the checksum A in Tx_1 will not match the content B in D.

This necessitates individual block checksums, since checksum mismatch of a single block is not a problem if the block belongs to a later valid transaction. In contrast, since the frozen state of metadata blocks are stored in the journal, checksumming over the entire set is sufficient for metadata transactional checksums. We explain how OptFS handles this during recovery shortly.

OptFS does not immediately write out checksummed data blocks; they are collected in memory and written in large batches upon transaction commit. This increases performance in some workloads.

5.3 Optimistic Techniques

We now describe the implementation of the optimistic journaling techniques. We also describe how OptFS reverts to more traditional mechanisms in some circumstances (*e.g.*, when the journal runs out of space).

In-Order Journal Recovery: OptFS recovers transactions in the order they were committed to the journal. A transaction can be replayed only if all its data blocks belong to valid transactions, and the checksum computed over metadata blocks matches that in the commit block.

OptFS performs recovery in two passes: the first pass linearly scans the journal, compiling a list of data blocks with checksum mismatches and the first journal transaction that contained each block. If a later valid transaction matches the block checksum, the block is deleted from the list. At the end of the scan, the earliest transaction in the list is noted. The next pass performs journal recovery until the faulting transaction, thus ensuring consistency.

OptFS journal recovery might take longer than ext4 recovery since OptFS might need to read data blocks off non-contiguous disk locations while ext4 only needs to read the contiguous journal to perform recovery.

In-Order Journal Release: When the virtual memory (VM) subsystem informs OptFS that checkpoint blocks have been acknowledged at time T, OptFS sets the transaction *cleanup* time as $T+T_D$, after which it is freed from the journal.

When the journal is running out of space, it may not be optimal to wait for the durability timeout interval before freeing up journal blocks. Under memory pressure, OptFS may need to free memory buffers of checkpoint blocks that have been issued to the disk and are waiting for durability timeouts. In such cases, OptFS issues a disk flush, ensuring the durability of checkpoint blocks that have been acknowledged by the disk. This allows OptFS to clean journal blocks belonging to some checkpointed transactions and free associated memory buffers.

Checksums: OptFS checksums data blocks using the same CRC32 algorithm used for metadata. A tag is created for each data block which stores the block number and its checksum. Data tags are stored in the descriptor blocks along with tags for metadata checksums.

Background Write after Notification: OptFS uses the VM subsystem to perform background writes. Checkpoint metadata blocks are marked as dirty and the `expiry` field of each block is set to be $T+T_D$ (the disk acknowledged the commit block at T). T will reflect the time that the entire transaction has been acknowledged because the commit is not issued until the disk acknowledges data and metadata writes. The blocks are then handed off to the VM subsystem.

During periodic background writes, the VM subsystem checks if each dirty block has expired: if so, the block is written out; otherwise the VM subsystem rechecks the block on its next periodic write-out cycle.

Reuse after Notification: Upon transaction commit, OptFS adds deleted data blocks to a global in-memory list of blocks that will be freed after the durability timeout, T_D. A background thread periodically frees blocks with expired durability timeouts. Upon file-system unmount, all list blocks are freed.

When the file system runs out of space, if the reuse list contains blocks, a disk flush is issued; this ensures the durability of transactions which freed the blocks in the list. These blocks are then set to be free. We expect that these "safety" flushes will be used infrequently.

Selective Data Journaling: Upon a block write, the block allocation information (which is reflected in `New` state of the `buffer_head`) is used to determine whether the block was newly allocated. If the write is an overwrite, the block is journaled as if it was metadata.

6 Evaluation

We now evaluate our prototype implementation of OptFS on two axes: reliability and performance. Experiments were performed on an Intel Core i3-2100 CPU with 16 GB of memory, a Hitachi DeskStar 7K1000.B 1 TB drive, and running Linux 3.2. The experiments were repeatable; numbers reported are the average over 10 runs.

Workload	Delayed Blocks	Crash points	
		Total	Consistent
Append	Data	50	50
	J_M, J_C	50	50
	Multiple blocks	100	100
Overwrite	Data	50	50
	J_M, J_C	50	50
	Multiple blocks	100	100

Table 1: **Reliability Evaluation.** *The table shows the total number of simulated crashpoints, and the number of crashpoints resulting in a consistent state after remount.*

6.1 Reliability

To verify OptFS's consistency guarantees, we build and apply a crash-robustness framework to it under two synthetic workloads. The first appends blocks to the end of a file; the second overwrites blocks of an existing file; both issue `osync()` calls frequently to induce more ordering points and thus stress OptFS machinery.

Crash simulation is performed by taking a workload trace, reordering some requests (either by delaying the write of a single data block, journal commit or metadata blocks, or multiple blocks chosen at random), creating a file-system image that contains a subset of those writes up until a particular crash point, and then recovering the file system from that image.

Table 1 shows the results of 400 different crash scenarios. OptFS recovers correctly in each case to a file system with a consistent prefix of updates (§4.4).

6.2 Performance

We now analyze the performance of OptFS under a number of micro- and macro-benchmarks. Figure 8 illustrates OptFS performance under these workloads; details of the workloads are found in the caption.

Micro-benchmarks: OptFS sequential-write performance is similar to ordered mode; however, sequential overwrites cause bandwidth to drop to half of that of ordered mode as OptFS writes every block twice. Random writes on OptFS are 3x faster than on ext4 ordered mode, as OptFS converts random overwrites into sequential journal writes (due to selective data journaling).

On the Createfiles benchmark, OptFS performs 2x better than ext4 ordered mode, as ext4 writes dirty blocks in the background for a number of reasons (*e.g.*, hitting the threshold for the amount of dirty in-memory data), while OptFS consolidates the writes and issues them upon commit. When we modified ext4 to stop the background writes, its performance was similar to OptFS.

Macro-benchmarks: We run the Filebench Fileserver, Webserver, and Varmail workloads [17]. OptFS performs similarly to ext4 ordered mode without flushes for Fileserver and Webserver. Varmail's frequent `fsync()` calls cause a significant number of flushes, leading to OptFS performing 7x better than ext4. ext4, even with flushes disabled, does not perform as well as OptFS since OptFS delays writing dirty blocks, issuing them in large batches periodically or on commit; in contrast, the background threads in ext4 issue writes in small batches so as to not affect foreground activity.

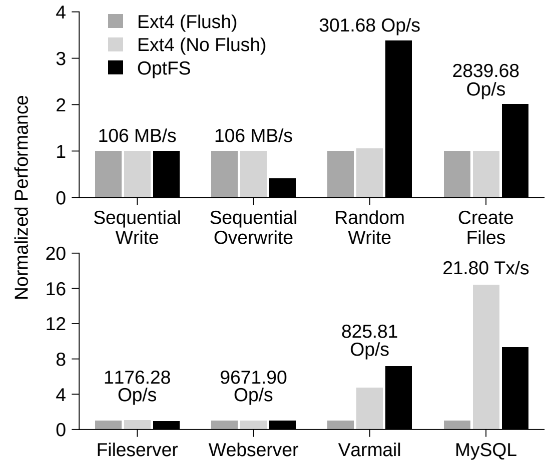

Figure 8: **Performance Comparison.** *Performance is shown normalized to ext4 ordered mode with flushes. The absolute performance of ordered mode with flushes is shown above each workload. Sequential writes are to 80 GB files. 200K random writes are performed over a 10 GB file, with an `fsync()` every 1K writes. The overwrite benchmark sequentially writes over a 32 GB file. Createfiles uses 64 threads to create 1M files. Fileserver emulates file-server activity, using 50 threads to perform a sequence of creates, deletes, appends, reads, and writes. Webserver emulates a multi-threaded web host server, performing sequences of open-read-close on multiple files plus a log file append, with 100 threads. Varmail emulates a multi-threaded mail server, performing a sequence of create-append-sync, read-append-sync, reads, and deletes in a single directory. Each workload was run for 660 seconds. MySQL OLTP benchmark performs 200K queries over a table with 1M rows.*

Finally, we run the MySQL OLTP benchmark from Sysbench [1] to investigate the performance on database workloads. OptFS performs 10x better than ordered mode with flushes, and 40% worse than ordered mode without flushes (due to the many in-place overwrites of MySQL, which result in selective data journaling).

Summary: OptFS significantly outperforms ordered mode with flushes on most workloads, providing the same level of consistency at considerably lower cost. On many workloads, OptFS performs as well as ordered mode without flushes, which offers no consistency guarantees. OptFS may not be suited for workloads which consist mainly of sequential overwrites.

6.3 Resource consumption

Table 2 compares the resource consumption of OptFS and ext4 for a 660 second run of Varmail. OptFS consumes 10% more CPU than ext4 ordered mode without flushes. Some of this overhead in our prototype can be

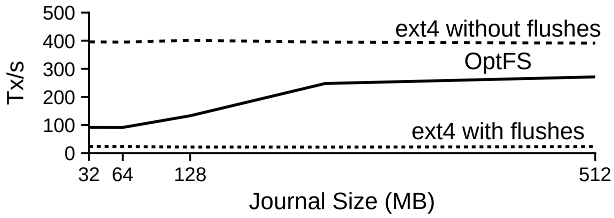

Figure 9: **Performance with Small Journals.** *The figure shows the variation in OptFS performance on the MySQL OLTP benchmark as the journal size is varied. When OptFS runs out of journal space, it issues flushes to safely checkpoint transactions. Note that even in this stress scenario, OptFS performance is 5x better than ext4 ordered mode with flushes.*

attributed to CRC32 data checksumming and the background thread which frees data blocks with expired durability timeouts, though further investigation is required.

OptFS delays checkpointing and holds metadata in memory longer, thereby increasing memory consumption. Moreover, OptFS delays data writes until transaction commit time, increasing performance for some workloads (*e.g.*, Filebench Createfiles), but at the cost of additional memory load.

6.4 Journal size

When OptFS runs out of journal space, it issues flushes in order to checkpoint transactions and free journal blocks. To investigate the performance of OptFS in such a situation, we reduce the journal size and run the MySQL OLTP benchmark. The results are shown in Figure 9. Note that due to selective journaling, OptFS performance will be lower than that of ext4 without flushes, even with large journals. We find that OptFS performs well with reasonably sized journals of 512 MB and greater; only with smaller journal sizes does performance degrade near the level of ext4 with flushes.

7 Case Studies

We now show how to use OptFS ordering guarantees to provide meaningful application-level crash consistency. Specifically, we study atomic updates in a text editor (gedit) and logging within a database (SQLite).

7.1 Atomic Update within Gedit

Many applications atomically update a file with the following sequence: first, create a new version of the file under a temporary name; second, call `fsync()` on the file to force it to disk; third, rename the file to the desired file name, replacing the original atomically with the new contents (some applications issue another `fsync()` to the parent directory, to persist the name change). The gedit text editor, which we study here, performs this sequence to update a file, ensuring that either the old or new contents are in place in their entirety, but never a mix.

File system	CPU %	Memory (MB)
ext4 ordered mode with flushes	3.39	486.75
ext4 ordered mode without flushes	14.86	516.03
OptFS	25.32	749.4

Table 2: **Resource Consumption.** *The table shows the average resource consumption by OptFS and ext4 ordered mode for a 660 second run of Filebench Varmail. OptFS incurs additional overhead due to techniques like delayed checkpointing.*

	gedit			SQLite		
	ext4 w/o flush	ext4 w/ flush	OptFS	ext4 w/o flush	ext4 w/ flush	OptFS
Total crashpoints	50	50	50	100	100	100
Inconsistent	7	0	0	73	0	0
Old state	26	21	36	8	50	76
New state	17	29	14	19	50	24
Time per op (ms)	1.12	39.4	0.84	23.38	152	15.3

Table 3: **Case Study: Gedit and SQLite.** *The table shows the number of simulated crashpoints that resulted in a consistent or inconsistent application state after remounting. It also shows the time required for an application operation.*

To study the effectiveness of OptFS for this usage, we modified gedit to use `osync()` instead of `fsync()`, thus ensuring order is preserved. We then take a block-level I/O trace when running under both ext4 (with and without flushes) and OptFS, and simulate a large number of crash points and I/O re-orderings in order to determine what happens during a crash. Specifically, we create a disk image that corresponds to a particular subset of writes taking place before a crash; we then mount the image, run recovery, and test for correctness.

Table 3 shows our results. With OptFS, the saved filename always points to either the old or new versions of the data in their entirety; atomic update is achieved. In a fair number of crashes, old contents are recovered, as OptFS delays writing updates; this is the basic trade-off OptFS makes, increasing performance but delaying durability. With ext4 (without flush), a significant number of crashpoints resulted in inconsistencies, including unmountable file systems and corrupt data. As expected, ext4 (with flush) did better, resulting in new or old contents exactly as dictated by the `fsync()` boundary.

The last row of Table 3 compares the performance of atomic updates in OptFS and ext4. OptFS delivers performance similar to ext4 without flushes, roughly 40x faster per operation than ext4 with flushes.

7.2 Temporary Logging in SQLite

We now investigate the use of `osync()` in a database management system, SQLite. To implement ACID transactions, SQLite first creates a temporary log file, writes some information to it, and calls `fsync()`. SQLite then updates the database file in place, calls `fsync()` on the database file, and finally deletes the log file. After a crash, if a log file is present, SQLite uses the log file for recovery (if necessary); the database is guaranteed to be recovered to either the pre- or post-transaction state.

Although SQLite transactions provide durability by default, its developers assert that many situations do not require it, and that "sync" can be replaced with pure ordering points in such cases. The following is an excerpt from the SQLite documentation [29]:

> As long as all writes that occur before the sync are completed before any write that happens after the sync, no database corruption will occur. [...] *the database will at least continue to be consistent, and that is what most people care about* (emphasis ours).

We now conduct the same consistency and performance tests for SQLite. With a small set of tables (≈30KB in size), we create a transaction to move records from one half the tables to the other half. After a simulated disk image that corresponds to a particular crashpoint is mounted, if consistent, SQLite should convert the database to either the pre-transaction (old) or post-transaction (new) state. The results in Table 3 are similar to the gedit case-study: OptFS always results in a consistent state, while ext4, without flushes, does not. OptFS performs 10x better than ext4 with flushes.

8 Related Work

A number of approaches to building higher performing file systems relate to our work on OptFS and optimistic crash consistency. For example, Soft Updates [11] shows how to carefully order disk updates so as to never leave an on-disk structure in an inconsistent form. In contrast with OptFS, FreeBSD Soft Updates issues flushes to implement `fsync()` and ordering (although the original work modified the SCSI driver to avoid issuing flushes).

Given the presence of asynchronous durability notifications, Soft Updates could be modified to take advantage of such signals. We believe doing so would be more challenging than modifying journaling file systems; while journaling works at the abstraction level of metadata and data, Soft Updates works directly with file-system structures, significantly increasing its complexity.

Our work is similar to that of Frost *et al.*'s work on Featherstitch [10], which provides a generalized framework to order file-system updates, in either a soft-updating or journal-based approach. Our work instead focuses on delayed ordering for journal commits; some of our techniques could increase the journal performance observed in their work.

Featherstitch provides similar primitives for ordering and durability: `pg_depend()` is similar to `osync()`, while `pg_sync()` is similar to `dsync()`. The main difference lies in the amount of work required from application developers: Featherstitch requires developers to explicitly encapsulate sets of file-system operations into units called *patchgroups* and define dependencies between them. Since `osync()` builds upon the familiar semantics of `fsync()`, we believe it will be easier for application developers to use.

Other work on "rethinking the sync" [19] has a similar flavor to our work. In that work, the authors cleverly note that disk writes only need to become durable when some external entity can observe said durability; thus, by delaying persistence until such externalization occurs, huge gains in performance can be realized. Our work is complimentary, in that it reduces the number of such durability events, instead enforcing a weaker (and higher performance) ordering among writes, but avoiding the complexity of implementing dependency tracking within the OS. In cases where durability is required (*i.e.*, applications use `dsync()` and not `osync()`), optimistic journaling does not provide much gain; thus, Nightingale *et al.*'s work still can be of benefit therein.

More recently, Chidambaram *et al.* implement NoFS [6], which removes the need for any ordering to disk at all, thus providing excellent performance. However, a lack of ordered writes means certain kinds of crashes can lead to a recovered file system that is consistent, but that contains data from partially completed operations. As a result, NoFS cannot implement atomic actions, such as `rename()`. Atomic rename is critical in the following update sequence: create temporary file; write temporary file with the entire contents of the old file, plus updates; persist temporary file via `fsync()`; atomically rename temporary file over old file. At the end of this sequence, the file should exist in either the old state or the new state with all the updates. If `rename()` is not atomic, or operations can be re-ordered, the entire file could be lost due to an inopportune crash (something that has been observed in deployment [9]). OptFS, in contrast, realizes many of the benefits of the NoFS approach but still delivers meaningful semantics upon which higher-level application semantics can be built.

Some real systems provide different flavors of `fsync()` to applications. For example, on Mac OS X, `fsync()` only ensures that writes have been issued to the disk from main memory; as the man page states, if an application desires durability, it should call `fcntl()` with the `F_FULLSYNC` flag. While the latter is identical to `dsync()`, the former is not equivalent to `osync()`; importantly, simply flushing a write to the disk cache does not provide any higher-level ordering guarantees and thus does not serve as a suitable primitive atop which to build application consistency; in contrast, as we have shown, `osync()` is quite apt for this usage scenario.

Prior work in distributed systems aims to measure eventual consistency (among replicas) and understand

Technique	Consistency	Performance	Availability	Durability	TX support	Flush-free	Optimistic
File-system check	L	H	L	L	×	√	√
Metadata journaling	M	M	H	H	√	×	×
Data journaling	H	M	H	H	√	×	×
Soft Updates	M	M	H	H	√	×	√
Copy-on-write	H	M	H	H	√	×	×
Backpointer-Based Consistency	M	H	H	L	×	√	√
Optimistic Crash Consistency	H	H	H	H*	√	√	√

Table 4: **Consistency Techniques.** *The table compares various approaches to providing consistency in file systems. Legend: L – Low, M – Medium, H – High. H* indicates that optimistic crash consistency can provide immediate durability, if required, using* `dsync()`*. Note that only optimistic crash consistency provides support for transactions, immediate durability on demand, and high performance while eliminating flushes in the common case.*

why it works in practice, similar to how probabilistic crash consistency seeks to understand how file systems maintain crash consistency without flushes. Yu *et al.* provide metrics to measure consistency among replicas and show that such quantification allows replicated services to make useful trade-offs [39]. Bailis *et al.* bound data staleness in eventually consistent distributed systems and explain why eventual consistency works in practice [2].

Finally, we compare optimistic crash consistency with other approaches to providing crash consistency in file systems. We have discussed some of them, such as Soft Updates and Backpointer-Based Consistency, in detail. Table 4 broadly compares various consistency techniques on aspects such as durability, availability, and performance. Transactional support indicates support for multi-block atomic operations such as `rename()`. Flush-free indicates that the technique does not issue flushes in the common case.

We observe that there are four approaches which are optimistic: the file-system check [5], Soft Updates, Backpointer-Based Consistency, and Optimistic Crash Consistency. Soft Updates issues flushes to ensure ordering, and hence is not flush-free. While the file-system check and backpointer-based consistency are completely flush-free, they cannot force writes to disk, and hence have eventual durability. Pessimistic approaches like journaling and copy-on-write employ flushes to provide transactional support and high levels of consistency and durability; however, they cannot implement ordering without flushes, and hence offer reduced performance on workloads with many ordering points. Due to `osync()` and `dsync()`, optimistic crash consistency can persist writes on demand (leading to immediate durability), and yet remain flush-free when only ordering is required.

9 Conclusion

We present optimistic crash consistency, a new approach to crash consistency in journaling file systems that uses a range of novel techniques to obtain *both* a high level of consistency and excellent performance in the common case. We introduce two new file-system primitives, `osync()` and `dsync()`, which decouple ordering from durability, and show that they can provide both high performance and meaningful semantics for application developers. We believe that such decoupling holds the key to resolving the constant tension between consistency and performance in file systems.

The source code for OptFS can be obtained at: `http://www.cs.wisc.edu/adsl/Software/optfs`. We hope that this will encourage adoption of the optimistic approach to consistency.

Acknowledgments

We thank James Mickens (our shepherd), the anonymous reviewers, and Eddie Kohler for their insightful comments. We thank Mark Hill, Michael Swift, Ric Wheeler, Joo-young Hwang, Sankaralingam Panneerselvam, Mohit Saxena, Asim Kadav, Arun Kumar, and the members of the ADSL lab for their feedback. We thank Sivasubramanian Ramasubramanian for helping us run the probabilistic crash consistency experiments. This material is based upon work supported by the NSF under CNS-1319405 and CNS-1218405 as well as donations from EMC, Facebook, Fusion-io, Google, Huawei, Microsoft, NetApp, Sony, and VMware. Any opinions, findings, and conclusions, or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF or other institutions.

References

[1] Alexey Kopytov. SysBench: a system performance benchmark. http://sysbench.sourceforge.net/index.html, 2004.

[2] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically Bounded Staleness for Practical Partial Quorums. *PVLDB*, 5(8):776–787, 2012.

[3] Steve Best. JFS Overview. http://jfs.sourceforge.net/project/pub/jfs.pdf, 2000.

[4] John S. Bucy, Jiri Schindler, Steven W. Schlosser, and Gregory R. Ganger. The DiskSim Simulation Environment Version 4.0 Reference Manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University, May 2008.

[5] Remy Card, Theodore Ts'o, and Stephen Tweedie. Design and Implementation of the Second Extended Filesystem. In *First Dutch International Symposium on Linux*, Amsterdam, Netherlands, December 1994.

[6] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Consistency Without Ordering. In *FAST '12*, pages 101–116, San Jose, CA, February 2012.

[7] James Cipar, Greg Ganger, Kimberly Keeton, Charles B Morrey III, Craig AN Soules, and Alistair Veitch. LazyBase: trading freshness for performance in a scalable database. In *EuroSys '12*, pages 169–182, Bern, Switzerland, April 2012.

[8] Jonathan Corbet. Barriers and Journaling Filesystems. http://lwn.net/Articles/283161, May 2008.

[9] Jonathan Corbet. That massive filesystem thread. http://lwn.net/Articles/326471/, March 2009.

[10] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized File System Dependencies. In *SOSP '07*, pages 307–320, Stevenson, WA, October 2007.

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

[12] Maurice Herlihy. Apologizing Versus Asking Permission: Optimistic Concurrency Control for Abstract Data Types. *ACM Transactions on Database Systems (TODS)*, 15(1):96–124, 1990.

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

[14] KnowledgeTek. Serial ATA Specification Rev. 3.0 Gold. http://www.knowledgetek.com/datastorage/courses/SATA_3.0-8.14.09(CD).pdf, 2009.

[15] Charles M. Kozierok. Overview and History of the SCSI Interface. http://www.pcguide.com/ref/hdd/if/scsi/over-c.html, 2001.

[16] Hsiang-Tsung Kung and John T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems (TODS)*, 6(2):213–226, 1981.

[17] Richard McDougall and Jim Mauro. Filebench. http://sourceforge.net/apps/mediawiki/filebench/index.php?title=Filebench, 2005.

[18] Marshall Kirk McKusick. Disks from the Perspective of a File System. *Communications of the ACM*, 55(11), 2012.

[19] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M Chen, and Jason Flinn. Rethink the Sync. In *OSDI '06*, pages 1–16, Seattle, WA, November 2006.

[20] Swapnil Patil, Anand Kashyap, Gopalan Sivathanu, and Erez Zadok. I^3FS: An In-kernel Integrity Checker and Intrusion detection File System. In *LISA '04*, pages 69–79, Atlanta, GA, November 2004.

[21] Zachary N. J. Peterson. Data Placement for Copy-on-write Using Virtual Contiguity. Master's thesis, U.C. Santa Cruz, 2002.

[22] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Analysis and Evolution of Journaling File Systems. In *USENIX '05*, pages 105–120, Anaheim, CA, April 2005.

[23] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *SOSP '05*, pages 206–220, Brighton, UK, October 2005.

[24] Margo Seltzer, Peter Chen, and John Ousterhout. Disk Scheduling Revisited. In *USENIX Winter '90*, pages 313–324, Washington, DC, January 1990.

[25] Margo I. Seltzer, Gregory R. Ganger, M. Kirk McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems. In *USENIX '00*, pages 71–84, San Diego, CA, June 2000.

[26] Dick Sites. How Fast Is My Disk? Systems Seminar at the University of Wisconsin-Madison, January 2013. http://www.cs.wisc.edu/event/how-fast-my-disk.

[27] David A. Solomon. *Inside Windows NT*. Microsoft Programming Series. Microsoft Press, 2nd edition, May 1998.

[28] Jon A. Solworth and Cyril U. Orji. Write-Only Disk Caches. In *SIGMOD '90*, pages 123–132, Atlantic City, NJ, May 1990.

[29] SQLite Team. How To Corrupt An SQLite Database File. http://www.sqlite.org/howtocorrupt.html, 2011.

[30] Marting Steigerwald. Imposing Order. Linux Magazine, May 2007.

[31] Christopher A. Stein, John H. Howard, and Margo I. Seltzer. Unifying File System Protection. In *USENIX '01*, pages 79–90, Boston, MA, June 2001.

[32] Adan Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the XFS File System. In *USENIX 1996*, San Diego, CA, January 1996.

[33] Theodore Tso. Re: [PATCH 0/4] (RESEND) ext3[34] barrier changes. Linux Kernel Mailing List. http://article.gmane.org/gmane.comp.file-systems.ext4/6662, May 2008.

[34] Theodore Ts'o and Stephen Tweedie. Future Directions for the Ext2/3 Filesystem. In *FREENIX '02*, Monterey, CA, June 2002.

[35] Stephen C. Tweedie. Journaling the Linux ext2fs File System. In *The Fourth Annual Linux Expo*, Durham, North Carolina, May 1998.

[36] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *NSDI '13*, pages 357–370, Lombard, IL, April 2013.

[37] Ralph O. Weber. SCSI Architecture Model - 3 (SAM-3). http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf, September 2004.

[38] B. L. Worthington, G. R. Ganger, and Y. N. Patt. Scheduling Algorithms for Modern Disk Drives. In *SIGMETRICS '94*, pages 241–251, Nashville, TN, May 1994.

[39] Haifeng Yu and Amin Vahdat. Design and Evaluation of a Continuous Consistency Model for Replicated Services. In *OSDI '00*, San Diego, CA, October 2000.