Journal-guided Resynchronization for Software RAID

Timothy E. Denehy, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau
Department of Computer Sciences, University of Wisconsin, Madison

Abstract

We investigate the problem of slow, scan-based, software RAID resynchronization that restores consistency after a system crash. Instead of augmenting the RAID layer to quicken the process, we leverage the functionality present in a journaling file system. We analyze Linux ext3 and introduce a new mode of operation, declared mode, that guarantees to provide a record of all outstanding writes in case of a crash. To utilize this information, we augment the software RAID interface with a verify read request, which repairs the redundant information for a block. The combination of these features allows us to provide fast, journal-guided resynchronization. We evaluate the effect of journal-guided resynchronization and find that it provides improved software RAID reliability and availability after a crash, while suffering little performance loss during normal operation.

1 Introduction

Providing reliability at the storage level often entails the use of RAID [8] to prevent data loss in the case of a disk failure. High-end storage arrays use specialized hardware to provide the utmost in performance and reliability [6]. Unfortunately, these solutions come with multi-million dollar price tags, and are therefore infeasible for many small to medium businesses and organizations.

Cost-conscious users must thus turn to commodity systems and a collection of disks to house their data. A popular, low-cost solution for reliability in this arena is software RAID [15], which is available on a range of platforms, including Linux, Solaris, FreeBSD, and Windows-based systems. This software-based approach is also attractive for specialized cluster-in-a-box systems. For instance, the EMC Centera [5] storage system is built from a cluster of commodity machines, each of which uses Linux software RAID to manage its disks.

Unfortunately, in life as in storage arrays, you get what you pay for. In the case of software RAID, the lack of non-volatile memory introduces a *consistent update* problem. Specifically, when a write is issued to the RAID layer, two (or more) disks must be updated in a consistent manner; the possibility of crashes makes this a challenge. For example, in a RAID-5 array, if an untimely crash occurs after the parity write completes but before the data block is written (*i.e.*, the two writes were issued in parallel but only one completed), the stripe is left in an inconsistent state. This inconsistency introduces a *window of vulnerability* – if a data disk fails before the stripe is made consistent, the data on that disk will be lost. Automatic reconstruction of the missing data block, based on the inconsistent parity, will silently return bad data to the client.

Hardware RAID circumvents this problem gracefully with non-volatile memory. By buffering an update in NVRAM until the disks have been consistently updated, a hardware-based approach avoids the window of vulnerability entirely. The outcome is ideal: both performance and reliability are excellent.

With current software-based RAID approaches, however, a performance/reliability trade-off must be made. Most current software RAID implementations choose performance over reliability [15]: they simply issue writes to the disks in parallel, hoping that an untimely crash does not occur in between. If a crash does occur, these systems employ an expensive *resynchronization* process: by scanning the entire volume, such discrepancies can be found and repaired. For large volumes, this process can take hours or even days.

The alternate software RAID approach chooses reliability over performance [3]. By applying write-ahead logging within the array to record the location of pending updates before they are issued, these systems avoid time-consuming resynchronization: during recovery, the RAID simply repairs the locations as recorded in its log. Unfortunately, removing the window of vulnerability comes with a high performance cost: each update within the RAID must now be preceded by a synchronous write to the log, greatly increasing the total I/O load on the disks.

To solve the consistent update problem within software RAID, and to develop a solution with both high performance and reliability, we take a global view of the storage stack: how can we leverage functionality within other layers of the system to assist us? In many cases, the client of the software RAID system will be a modern journaling file system, such as the default Linux file system, ext3 [16, 17, 18], or ReiserFS [9], JFS [1], or Windows NTFS [12]. Although standard journaling techniques maintain the consistency of file system data structures, they do not solve the consistent update problem at the RAID level. We find, however, that journaling can be readily augmented to do so.

Specifically, we introduce a new mode of operation within Linux ext3: *declared mode*. Before writing to any permanent locations, declared mode records its intentions in the file system journal. This functionality guarantees a record of all outstanding writes in the event of a crash. By consulting this activity record, the file system knows which blocks were in the midst of being updated and hence can dramatically reduce the window of vulnerability following a crash.

To complete the process, the file system must be able to communicate its information about possible vulnerabilities to the RAID layer below. For this purpose, we add a new interface to the software RAID layer: the *verify read*. Upon receiving a verify read request, the RAID layer reads the requested block as well as its mirror or parity group and verifies the redundant information. If an irregularity is found, the RAID layer re-writes the mirror or parity to produce a consistent state.

We combine these features to integrate journal-guided resynchronization into the file system recovery process. Using our record of write activity vastly decreases the time needed for resynchronization, in some cases from a period of days to mere seconds. Hence, our approach avoids the performance/reliability trade-off found in software RAID systems: performance remains high and the window of vulnerability is greatly reduced.

In general, we believe the key to our solution is its *cooperative* nature. By removing the strict isolation between the file system above and the software RAID layer below, these two subsystems can work *together* to solve the consistent update problem without sacrificing either performance or reliability.

The rest of the paper is organized as follows. Section 2 illustrates the software RAID consistent update problem and quantifies the likelihood that a crash will lead to data vulnerability. Section 3 provides an introduction to the ext3 file system and its operation. In Section 4, we analyze ext3's write activity, introduce ext3 declared mode and an addition to the software RAID interface, and merge RAID resynchronization into the journal recovery process. Section 5 evaluates the performance of declared mode and the effectiveness of journal-guided resynchronization. We discuss related work in Section 6, and conclude in Section 7.

2 The Consistent Update Problem

2.1 Introduction

The task of a RAID is to maintain an invariant between the data and the redundant information it stores. These invariants provide the ability to recover data in the case of a disk failure. For RAID-1, this means that each mirrored block contains the same data. For parity schemes, such as RAID-5, this means that the parity block for each stripe stores the exclusive-or of its associated data blocks.

However, because the blocks reside on more than one disk, updates cannot be applied atomically. Hence, maintaining these invariants in the face of failure is challenging. If a crash occurs during a write to an array, its blocks may be left in an inconsistent state. Perhaps only one mirror was successfully written to disk, or a data block may have been written without its parity update.

We note here that the consistent update problem and its solutions are distinct from the traditional problem of RAID disk failures. When such a failure occurs, all of the redundant information in the array is lost, and thus all of the data is vulnerable to a second disk failure. This situation is solved by the process of reconstruction, which regenerates all of the data located on the failed disk.

2.2 Failure Models

We illustrate the consistent update problem with the example shown in Figure 1. The diagram depicts the state of a single stripe of blocks from a four disk RAID-5 array as time progresses from left to right. The software RAID layer residing on the machine is servicing a write to data block **Z**, and it must also update the parity block, **P**. The machine issues the data block write at time 1, it is written to disk at time 3, and the machine is notified of its completion at time 4. Similarly, the parity block is issued at time 2, written at time 5, and its notification arrives at time 6. After the data write to block **Z** at time 3, the stripe enters a *window of vulnerability*, denoted by the shaded blocks. During this time, the failure of any of the first three disks will result in data loss. Because the stripe's data and parity blocks exist in an inconsistent state, the data residing on a failed disk cannot be reconstructed. This inconsistency is corrected at time 5 by the write to **P**.

We consider two failure models to allow for the possibility of independent failures between the host machine

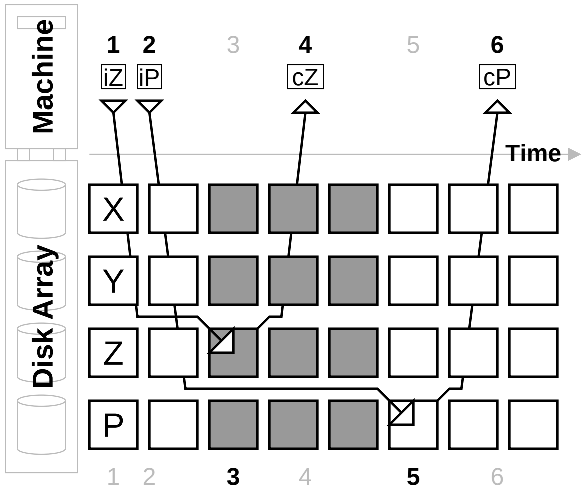


Figure 1: **Failure Scenarios.** *The diagram illustrates the sequence of events for a data block write and a parity update to a four disk RAID-5 array as time progresses from left to right. The boxes labeled* i *indicate a request being issued, and those labeled* c *represent completions. The shaded blocks denote a window of vulnerability.*

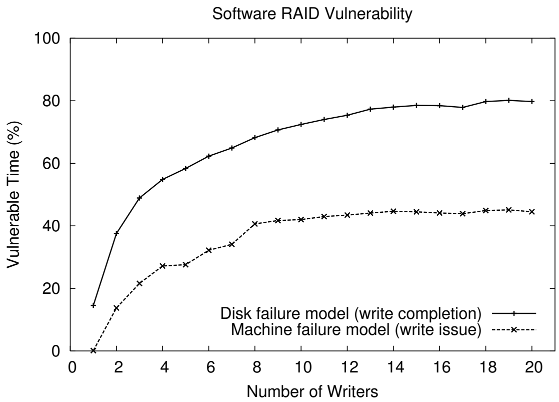


Figure 2: **Software RAID Vulnerability.** *The graph plots the percent of time (over the duration of the experiment) that an inconsistent disk state exists in the RAID-5 array as the number of writers increases along the x-axis. Vulnerabilities due to disk failure and machine failure are plotted separately.*

and the array of disks. We will discuss each in turn and relate their consequences to the example in Figure 1. The *machine failure model* includes events such as operating system crashes and machine power losses. In our example, if the machine crashes between times 1 and 2, and the array remains active, the stripe will be left in an inconsistent state after the write completes at time 3.

Our second model, the *disk failure model*, considers power losses at the disk array. If such a failure occurs between time 3 and time 5 in our example, the stripe will be left in a vulnerable state. Note that the disk failure model encompasses non-independent failures such as a simultaneous power loss to the machine and the disks.

2.3 Measuring Vulnerability

To determine how often a crash or failure could leave an array in an inconsistent state, we instrument the Linux software RAID-5 layer and the SCSI driver to track several statistics. First, we record the amount of time between the first write issued for a stripe and the last write issued for a stripe. This measures the difference between times 1 and 2 in Figure 1, and corresponds directly to the period of vulnerability under the machine failure model.

Second, we record the amount of time between the first write completion for a stripe and the last write completion for a stripe. This measures the difference between time 4 and time 6 in our example. Note, however, that the vulnerability under the disk failure model occurs between time 3 and time 5, so our measurement is an approximation. Our results may slightly overestimate or underestimate the actual vulnerability depending on the time it takes each completion to be sent to and processed by the host machine. Finally, we track the number of stripes that are vulnerable for each of the models. This allows us to calculate the percent of time that any stripe in the array is vulnerable to either type of failure.

Our test workload consists of multiple threads performing synchronous, random writes to a set of files on the array. All of our experiments are performed on an Intel Pentium Xeon 2.6 GHz processor with 512 MB of RAM running Linux kernel 2.6.11. The machine has five IBM 9LZX disks configured as a 1 GB software RAID-5 array. The RAID volume is sufficiently large to perform our benchmarks yet small enough to reduce the execution time of our resynchronization experiments.

Figure 2 plots the percent of time (over the duration of the experiment) that any array stripe is vulnerable as the number of writers in the workload is increased along the x-axis. As expected, the cumulative window of vulnerability increases as the amount of concurrency in the workload is increased. The vulnerability under the disk failure model is greater because it is dependent on the response time of the write requests. Even for a small number of writers, it is more than likely that a disk failure will result in an inconsistent state. For higher concurrency, the array exists in a vulnerable state for up to 80% of the length of the experiment.

The period of vulnerability under the machine failure model is lower because it depends only on the processing time needed to issue the write requests. In our experiment, vulnerability reaches approximately 40%. At much higher concurrencies, however, the ability to issue requests could be impeded by full disk queues. In this case, the machine vulnerability will also depend on the disk response time and will increase accordingly.

2.4 Solutions

To solve this problem, high-end RAID systems make use of non-volatile storage, such as NVRAM. When a write request is received, a log of the request and the data are first written to NVRAM, and then the updates are propagated to the disks. In the event of a crash, the log records and data present in the NVRAM can be used to replay the writes to disk, thus ensuring a consistent state across the array. This functionality comes at an expense, not only in terms of raw hardware, but in the cost of developing and testing a more complex system.

Software RAID, on the other hand, is frequently employed in commodity systems that lack non-volatile storage. When such a system reboots from a crash, there is no record of write activity in the array, and therefore no indication of where RAID inconsistencies may exist. Linux software RAID rectifies this situation by laboriously reading the contents of the entire array, checking the redundant information, and correcting any discrepancies. For RAID-1, this means reading both data mirrors, comparing their contents, and updating one if their states differ. Under a RAID-5 scheme, each stripe of data must be read and its parity calculated, checked against the parity on disk, and re-written if it is incorrect.

This approach fundamentally affects both reliability and availability. The time-consuming process of scanning the entire array lengthens the window of vulnerability during which inconsistent redundancy may lead to data loss under a disk failure. Additionally, the disk bandwidth devoted to resynchronization has a deleterious effect on the foreground traffic serviced by the array. Consequently, there exists a fundamental tension between the demands of reliability and availability: allocating more bandwidth to recover inconsistent disk state reduces the availability of foreground services, but giving preference to foreground requests increases the time to resynchronize.

As observed by Brown and Patterson [2], the default Linux policy addresses this trade-off by favoring availability over reliability, limiting resynchronization bandwidth to 1000 KB/s per disk. Unfortunately, such a slow rate may equate to days of repair time and vulnerability for even moderately sized arrays of hundreds of gigabytes. Figure 3 illustrates this problem by plotting an analytical model of the resynchronization time for a five disk array as the raw size of the array increases along the x-axis. With five disks, the default Linux policy will take almost four minutes of time to scan and repair each gigabyte of disk space, which equates to *two and a half days* for a terabyte of capacity. Disregarding the availability of the array, even modern interconnects would need approximately an hour at their full bandwidth to resynchronize the same one terabyte array.

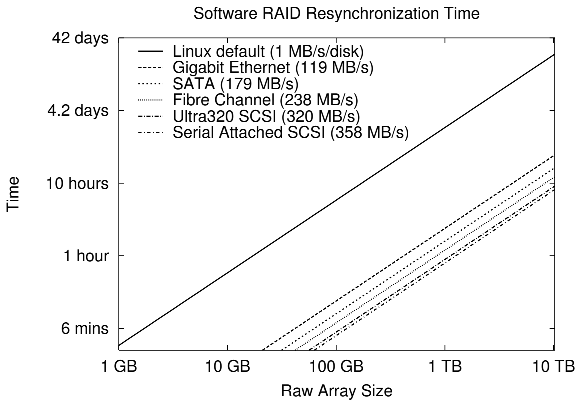


Figure 3: **Software RAID Resynchronization Time.** *The graph plots the time to resynchronize a five disk array as the raw capacity increases along the x-axis.*

One possible solution to this problem is to add logging to the software RAID system in a manner similar to that discussed above. This approach suffers from two drawbacks, however. First, logging to the array disks themselves would likely decrease the overall performance of the array by interfering with foreground requests. The high-end solution discussed previously benefits from fast, independent storage in the form of NVRAM. Second, adding logging and maintaining an acceptable level of performance could add considerable complexity to the software. For instance, the Linux software RAID implementation uses little buffering, discarding stripes when their operations are complete. A logging solution, however, may need to buffer requests significantly in order to batch updates to the log and improve performance.

Another solution is to perform intent logging to a bitmap representing regions of the array. This mechanism is used by the Solaris Volume Manager [14] and the Veritas Volume Manager [19] to provide optimized resynchronization. An implementation for Linux software RAID-1 is also in development [3], though it has not been merged into the main kernel. Like logging to the array, this approach is likely to suffer from poor performance. For instance, the Linux implementation performs a synchronous write to the bitmap before updating data in the array to ensure proper resynchronization. Performance may be improved by increasing the bitmap granularity, but this comes at the cost of performing scan-based resynchronization over larger regions.

Software RAID is just one layer in the storage hierarchy. One likely configuration contains a modern journaling file system in the layer above, logging disk updates to maintain consistency across its on-disk data structures. In the next sections, we examine how a journaling file system can be used to solve the software RAID resynchronization problem.

3 ext3 Background

In this section, we discuss the Linux ext3 file system, its operation, and its data structures. These details will be useful in our analysis of its write activity and the description of our modifications to support journal-guided resynchronization in Section 4. Although we focus on ext3, we believe our techniques are general enough to apply to other journaling file systems, such as ReiserFS and JFS for Linux, and NTFS for Windows.

Linux ext3 is a modern journaling file system that aims to keep complex on-disk data structures in a consistent state. To do so, all file system updates are first written to a log called the journal. Once the journal records are stored safely on disk, the updates can be applied to their home locations in the main portion of the file system. After the updates are propagated, the journal records are erased and the space they occupied can be re-used.

This mechanism greatly improves the efficiency of crash recovery. After a crash, the journal is scanned and outstanding updates are replayed to bring the file system into a consistent state. This approach constitutes a vast improvement over the previous process (*i.e.* fsck [7]) that relied on a full scan of the file system data structures to ensure consistency. It seems natural, then, to make use of the same journaling mechanism to improve the process of RAID resynchronization after a crash.

3.1 Modes

The ext3 file system offers three modes of operation: data-journaling mode, ordered mode, and writeback mode. In data-journaling mode, all data and metadata is written to the journal, coordinating all updates to the file system. This provides very strong consistency semantics, but at the highest cost. All data written to the file system is written twice: first to the journal, then to its home location.

Ordered mode, the ext3 default, writes all file system metadata to the journal, but file data is written directly to its home location. In addition, this mode guarantees a strict ordering between the writes: all file data for a transaction is written to disk before the corresponding metadata is written to the journal and committed. This guarantees that file metadata will never reference a data block before it has been written. Thus, this mechanism provides the same strong consistency as data-journaling mode without the expense of multiple writes for file data.

In writeback mode, only file system metadata is written to the journal. Like ordered mode, file data is written directly to its home location; unlike ordered mode, however, writeback mode provides no ordering guarantees between metadata and data, therefore offering much weaker consistency. For instance, the metadata for a file creation may be committed to the journal before the file data is written. In the event of a crash, journal recovery will restore the file metadata, but its contents could be filled with arbitrary data. We will not consider writeback mode for our purposes because of its weaker consistency and its lack of write ordering.

3.2 Transaction Details

To reduce the overhead of file system updates, sets of changes are grouped together into compound transactions. These transactions exist in several phases over their lifetimes. Transactions start in the *running* state. All file system data and metadata updates are associated with the current running transaction, and the buffers involved in the changes are linked to the in-memory transaction data structure. In ordered mode, data associated with the running transaction may be written at any time by the kernel pdflush daemon, which is responsible for cleaning dirty buffers. Periodically, the running transaction is closed and a new transaction is started. This may occur due to a timeout, a synchronization request, or because the transaction has reached a maximum size.

Next, the closed transaction enters the *commit* phase. All of its associated buffers are written to disk, either to their home locations or to the journal. After all of the transaction records reside safely in the journal, the transaction moves to the *checkpoint* phase, and its data and metadata are copied from the journal to their permanent home locations. If a crash occurs before or during the checkpoint of a committed transaction, it will be checkpointed again during the journal *recovery* phase of mounting the file system. When the checkpoint phase completes, the transaction is removed from the journal and its space is reclaimed.

3.3 Journal Structure

Tracking the contents of the journal requires several new file system structures. A journal superblock stores the size of the journal file, pointers to the head and tail of the journal, and the sequence number of the next expected transaction. Within the journal, each transaction begins with a *descriptor* block that lists the permanent block addresses for each of the subsequent data or metadata blocks. More than one descriptor block may be needed depending on the number of blocks involved in a transaction. Finally, a *commit* block signifies the end of a particular transaction. Both descriptor blocks and commit blocks begin with a magic header and a sequence number to identify their associated transaction.

4 Design and Implementation

The goal of resynchronization is to correct any RAID inconsistencies that result from system crash or failure. If we can identify the outstanding write requests at the time of the crash, we can significantly narrow the range of blocks that must be inspected. This will result in faster resynchronization and improved reliability and availability. Our hope is to recover such a record of outstanding writes from the file system journal. To this end, we begin by examining the write activity generated by each phase of an ext3 transaction.

4.1 ext3 Write Analysis

In this section, we examine each of the ext3 transaction operations in detail. We emphasize the write requests generated in each phase, and we characterize the possible disk states resulting from a crash. Specifically, we classify each write request as targeting a known location, an unknown location, or a bounded location, based on its record of activity in the journal. Our goal, upon restarting from a system failure, is to recover a record of *all outstanding write requests* at the time of the crash.

Running:

1. In ext3 ordered mode, the pdflush daemon may write dirty pages to disk while the transaction is in the running state. If a crash occurs in this state, the affected locations will be unknown, as *no record of the ongoing writes will exist in the journal.*

Commit:

1. ext3 writes all un-journaled dirty data blocks associated with the transaction to their home locations, and waits for the I/O to complete. This step applies only to ordered mode, since all data in data-journaling mode is destined for the journal. If a crash occurs during this phase, the locations of any outstanding writes will be unknown.

2. ext3 writes descriptors, journaled data, and metadata blocks to the journal, and waits for the writes to complete. In ordered mode, only metadata blocks will be written to the journal, whereas all blocks are written to the journal in data-journaling mode. If the system fails during this phase, no specific record of the ongoing writes will exist, but all of the writes will be bounded within the fixed location journal.

3. ext3 writes the transaction commit block to the journal, and waits for its completion. In the event of a crash, the outstanding write is again bounded within the journal.

Block Type	Data-journaling Mode
superblock	known, fixed location
journal	bounded, fixed location
home metadata	known, journal descriptors
home data	known, journal descriptors

Block Type	Ordered Mode
superblock	known, fixed location
journal	bounded, fixed location
home metadata	known, journal descriptors
home data	unknown

Table 1: **Journal Write Records.** *The table lists the block types written during transaction processing and how their locations can be determined after a crash.*

Checkpoint:

1. ext3 writes journaled blocks to their home locations and waits for the I/O to complete. If the system crashes during this phase, the ongoing writes can be determined from the descriptor blocks in the journal, and hence they affect known locations.

2. ext3 updates the journal tail pointer in the superblock to signify completion of the checkpointed transaction. A crash during this operation involves an outstanding write to the journal superblock, which resides in a known, fixed location.

Recovery:

1. ext3 scans the journal checking for the expected transaction sequence numbers (based on the sequence in the journal superblock) and records the last committed transaction.

2. ext3 checkpoints each of the committed transactions in the journal, following the steps specified above. All write activity occurs to known locations.

Table 1 summarizes our ability to locate ongoing writes after a crash for the data-journaling and ordered modes of ext3. In the case of data-journaling mode, the locations of any outstanding writes can be determined (or at least bounded) during crash recovery, be it from the journal descriptor blocks or from the fixed location of the journal file and superblock. Thus, the existing ext3 data-journaling mode is quite amenable to assisting with the problem of RAID resynchronization. On the down side, however, data-journaling typically provides the least performance of the ext3 family.

For ext3 ordered mode, on the other hand, data writes to permanent home locations are not recorded in the journal data structures, and therefore cannot be located during crash recovery. We now address this deficiency with a modified ext3 ordered mode: declared mode.

4.2 ext3 Declared Mode

In the previous section we concluded that, if a crash occurs while writing data directly to its permanent location, the ext3 ordered mode journal will contain no record of those outstanding writes. The locations of any RAID level inconsistencies caused by those writes will remain unknown upon restart. To overcome this deficiency, we introduce a new variant of ordered mode, *declared mode*.

Declared mode differs from ordered mode in one key way: it guarantees that a write record for each data block resides safely in the journal before that location is modified. Effectively, the file system must *declare its intent* to write to any permanent location before issuing the write.

To keep track of these intentions, we introduce a new journal block, the *declare* block. A set of declare blocks is written to the journal at the beginning of each transaction commit phase. Collectively, they contain a list of all permanent locations to which data blocks in the transaction will be written. Though their construction is similar to that of descriptor blocks, their purpose is quite different. Descriptor blocks list the permanent locations for blocks that appear in the journal, whereas declare blocks list the locations of blocks that *do not appear* in the journal. Like descriptor and commit blocks, declare blocks begin with a magic header and a transaction sequence number. Declared mode thus adds a single step to the beginning of the commit phase, which proceeds as follows:

Declared Commit:

1. ext3 writes declare blocks to the journal listing each of the permanent data locations to be written as part of the transaction, and it waits for their completion.

2. ext3 writes all un-journaled data blocks associated with the transaction to their home locations, and waits for the I/O to complete.

3. ext3 writes descriptors and metadata blocks to the journal, and waits for the writes to complete.

4. ext3 writes the transaction commit block to the journal, and waits for its completion.

The declare blocks at the beginning of each transaction introduce an additional space cost in the journal. This cost varies with the number of data blocks each transaction contains. In the best case, one declare block will be added for every 506 data blocks, for a space overhead of 0.2%. In the worst case, however, one declare block will be needed for a transaction containing only a single data block. We investigate the performance consequences of these overheads in Section 5.

Implementing declared mode in Linux requires two main changes. First, we must guarantee that no data buffers are written to disk before they have been declared in the journal. To accomplish this, we refrain from setting the dirty bit on modified pages managed by the file system. This prevents the pdflush daemon from eagerly writing the buffers to disk during the running state. The same mechanism is used for all metadata buffers and for data buffers in data-journaling mode, ensuring that they are not written before they are written to the journal.

Second, we need to track data buffers that require declarations, and write their necessary declare blocks at the beginning of each transaction. We start by adding a new *declare tree* to the in-memory transaction structure, and ensure that all declared mode data buffers are placed on this tree instead of the existing *data list*. At the beginning of the commit phase, we construct a set of declare blocks for all of the buffers on the declare tree and write them to the journal. After the writes complete, we simply move all of the buffers from the declare tree to the existing transaction data list. The use of a tree ensures that the writes occur in a more efficient order, sorted by block address. From this point, the commit phase can continue without modification. This implementation minimizes the changes to the shared commit procedure; the other ext3 modes simply bypass the empty declare tree.

4.3 Software RAID Interface

Initiating resynchronization at the file system level requires a mechanism to repair suspected inconsistencies after a crash. A viable option for RAID-1 arrays is for the file system to read and re-write any blocks it has deemed vulnerable. In the case of inconsistent mirrors, either the newly written data or the old data will be restored to each block. This achieves the same results as the current RAID-1 resynchronization process. Because the RAID-1 layer imposes no ordering on mirrored updates, it cannot differentiate new data from old data, and merely chooses one block copy to restore consistency.

This read and re-write strategy is unsuitable for RAID-5, however. When the file system re-writes a single block, our desired behavior is for the RAID layer to calculate its parity across the entire stripe of data. Instead, the RAID layer could perform a read-modify-write by reading the target block and its parity, re-calculating the parity, and writing both blocks to disk. This operation depends on the consistency of the data and parity blocks it reads from disk. If they are not consistent, it will produce incorrect results, simply prolonging the discrepancy. In general, then, a new interface is required for the file system to communicate possible inconsistencies to the software RAID layer.

We consider two options for the new interface. The first requires the file system to read each vulnerable block and then re-write it with an explicit *reconstruct write* request. In this option, the RAID layer is responsible for reading the remainder of the block's parity group, re-calculating its parity, and then writing the block and the new parity to disk. We are dissuaded from this option because it may perform unnecessary writes to consistent stripes that could cause further vulnerabilities in the event of another crash.

Instead, we opt to add an explicit *verify read* request to the software RAID interface. In this case, the RAID layer reads the requested block along with the rest of its stripe and checks to make sure that the parity is consistent. If it is not, the newly calculated parity is written to disk to correct the problem.

The Linux implementation for the verify read request is rather straight-forward. When the file system wishes to perform a verify read request, it marks the corresponding buffer head with a new *RAID synchronize* flag. Upon receiving the request, the software RAID-5 layer identifies the flag and enables an existing *synchronizing* bit for the corresponding stripe. This bit is used to perform the existing resynchronization process. Its presence causes a read of the entire stripe followed by a parity check, exactly the functionality required by the verify read request.

Finally, an option is added to the software RAID-5 layer to disable resynchronization after a crash. This is our most significant modification to the strict layering of the storage stack. The RAID module is asked to entrust its functionality to another component for the overall good of the system. Instead, an apprehensive software RAID implementation may delay its own efforts in hopes of receiving the necessary verify read requests from the file system above. If no such requests arrive, it could start its own resynchronization to ensure the integrity of its data and parity blocks.

4.4 Recovery and Resynchronization

Using ext3 in either data-journaling mode or declared mode guarantees an accurate view of all outstanding write requests at the time of a crash. Upon restart, we utilize this information and our verify read interface to perform fast, file system guided resynchronization for the RAID layer. Because we make use of the file system journal, and because of ordering constraints between their operations, we combine this process with journal recovery. The dual process of file system recovery and RAID resynchronization proceeds as follows:

Recovery and Resync:

1. ext3 performs verify reads for its superblock and the journal superblock, ensuring their consistency in case they were being written during the crash.

2. ext3 scans the journal checking for the expected transaction sequence numbers (based on the sequence in the journal superblock) and records the last committed transaction.

3. For the first committed transaction in the journal, ext3 performs verify reads for the home locations listed in its descriptor blocks. This ensures the integrity of any blocks undergoing checkpoint writes at the time of the crash. Only the first transaction need be examined because checkpoints must occur in order, and each checkpointed transaction is removed from the journal before the next is processed. Note that these verify reads must take place before the writes are replayed below to guarantee the parity is up-to-date. Adding the explicit reconstruct write interface mentioned earlier would negate the need for this two step process.

4. ext3 issues verify reads beyond the last committed transaction (at the head of the journal) for the length of the maximum transaction size. This corrects any inconsistent blocks as a result of writing the next transaction to the journal.

5. While reading ahead in the journal, ext3 identifies any declare blocks and descriptor blocks for the next uncommitted transaction. If no descriptor blocks are found, it performs verify reads for the permanent addresses listed in each declare block, correcting any data writes that were outstanding at the time of the crash. Declare blocks from transactions containing descriptors can be ignored, as their presence constitutes evidence for the completion of all data writes to permanent locations.

6. ext3 checkpoints each of the committed transactions in the journal as described in Section 4.1.

The implementation re-uses much of the existing framework for the journal recovery process. Issuing the necessary verify reads means simply adding the RAID synchronize flag to the buffers already used for reading the journal or replaying blocks. The verify reads for locations listed in descriptor blocks are handled as the replay writes are processed. The journal verify reads and declare block processing for an uncommitted transaction are performed after the final pass of the journal recovery.

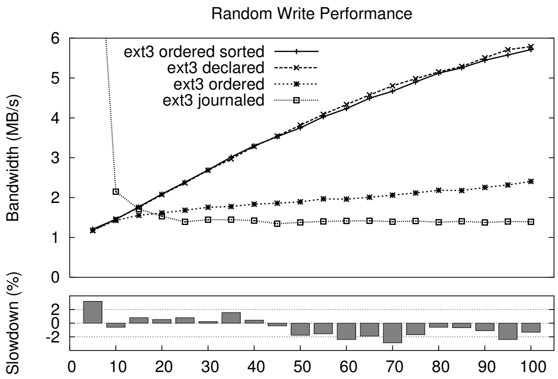


Figure 4: **Random Write Performance.** *The top graph plots random write performance as the amount of data written is increased along the x-axis. Data-journaling mode achieves 11.07 MB/s when writing 5 MB of data. The bottom graph shows the relative performance of declared mode as compared to ordered mode with sorting.*

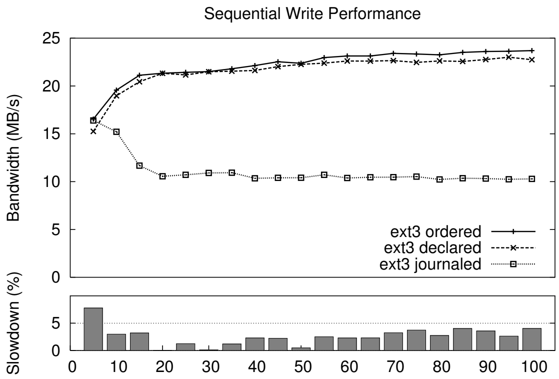


Figure 5: **Sequential Write Performance.** *The top graph plots sequential write performance as the amount of data written is increased along the x-axis. The bottom graph shows the relative performance of declared mode as compared to ordered mode.*

5 Evaluation

In this section, we evaluate the performance of ext3 declared mode and compare it to ordered mode and data-journaling mode. We hope that declared mode adds little overhead despite writing extra declare blocks for each transaction. After our performance evaluation, we examine the effects of journal-guided resynchronization. We expect that it will greatly reduce resync time and increase available bandwidth for foreground applications. Finally, we examine the complexity of our implementation.

5.1 ext3 Declared Mode

We begin our performance evaluation of ext3 declared mode with two microbenchmarks, random write and sequential write. First, we test the performance of random writes to an existing 100 MB file. A call to fsync() is used at the end of the experiment to ensure that all data reaches disk. Figure 4 plots the bandwidth achieved by each ext3 mode as the amount written is increased along the x-axis. All of our graphs plot the mean of five experimental trials.

We identify two points of interest on the graph. First, data-journaling mode underperforms ordered mode as the amount written increases. Note that data-journaling mode achieves 11.07 MB/s when writing only 5 MB of data because the random write stream is transformed into a large sequential write that fits within the journal. As the amount of data written increases, it outgrows the size of the journal. Consequently, the performance of data-journaling decreases because each block is written twice, first to the journal, and then to its home location. Ordered mode garners better performance by writing data directly to its permanent location.

Second, we find that declared mode greatly outperforms ordered mode as the amount written increases. Tracing the disk activity of ordered mode reveals that part of the data is issued to disk in sorted order based on walking the dirty page tree. The remainder, however, is issued unsorted by the commit phase as it attempts to complete all data writes for the transaction. Adding sorting to the commit phase of ordered mode solves this problem, as evidenced by the performance plotted in the graph. The rest of our performance evaluations are based on this modified version of ext3 ordered mode with sorted writing during commit.

Finally, the bottom graph in Figure 4 shows the slowdown of declared mode relative to ordered mode (with sorting). Overall, the performance of the two modes is extremely close, differing by no more than 3.2%.

Our next experiment tests sequential write performance to an existing 100 MB file. Figure 5 plots the performance of the three ext3 modes. Again, the amount written is increased along the x-axis, and fsync() is used to ensure that all data reaches disk. Ordered mode and declared mode greatly outperform data-journaling mode, achieving 22 to 23 MB/s compared to just 10 MB/s.

The bottom graph in Figure 5 shows the slowdown of ext3 declared mode as compared to ext3 ordered mode. Declared mode performs quite well, within 5% of ordered mode for most data points. Disk traces reveal that the performance loss is due to the fact that declared mode waits for fsync() to begin writing declare blocks and data. Because of this, ordered mode begins writing data to disk slightly earlier than declared mode. To alleviate this delay, we implement an early declare mode that begins writing declare blocks to the journal as soon as possible, that is, as soon as enough data blocks have been modified to fill a declare block. Unfortunately, this

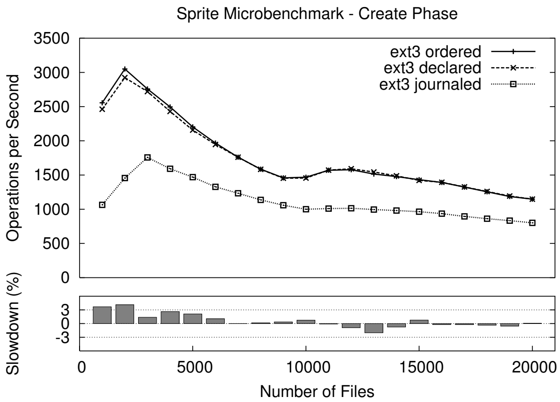


Figure 6: **Sprite Create Performance.** *The top graph plots the performance of the create phase of the Sprite LFS microbenchmark as the number of files increases along the x-axis. The bottom graph shows the slowdown of declared mode when compared to ordered mode.*

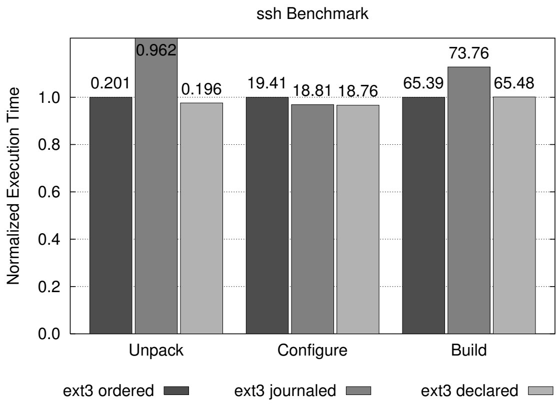


Figure 7: **ssh Benchmark Performance.** *The graph plots the normalized execution time of the unpack, configure, and build phases of the ssh benchmark as compared to ext3 ordered mode. The absolute execution times in seconds are listed above each bar.*

modification does not result in a performance improvement. The early writing of a few declare blocks and data blocks is offset by the seek activity between the journal and the home data locations (not shown).

Next, we examine the performance under the Sprite LFS microbenchmark [10], which creates, reads, and then unlinks a specified number of 4 KB files. Figure 6 plots the number of create operations completed per second as the number of files is increased along the x-axis. The bottom graph shows the slowdown of declared mode relative to ordered mode. Declared mode performs well, within 4% of ordered mode for all cases. The performance of declared mode and ordered mode are nearly identical for the other phases of the benchmark.

The ssh benchmark unpacks, configures, and builds version 2.4.0 of the ssh program from a tarred and compressed distribution file. Figure 7 plots the performance of each mode during the three stages of the benchmark. The execution time of each stage is normalized to that of ext3 ordered mode, and the absolute times in seconds are listed above each bar. Data-journaling mode is slighter faster than ordered mode for the configure phase, but it is 12% slower during build and 378% slower during unpack. Declared mode is quite comparable to ordered mode, running about 3% faster during unpack and configure, and 0.1% slower for the build phase.

Next, we examine ext3 performance on a modified version of the postmark benchmark that creates 5000 files across 71 directories, performs a specified number of transactions, and then deletes all files and directories. Our modification involves the addition of a call to sync() after each phase of the benchmark to ensure that data is written to disk. The unmodified version exhibits unusually high variances for all three modes of operation.

The execution time for the benchmark is shown in Figure 8 as the number of transactions increases along the x-axis. Data-journaling mode is extremely slow, and therefore we concentrate on the other two modes, for which we identify two interesting points. First, for large numbers of transactions, declared mode compares favorably to ordered mode, differing by approximately 5% in the worst cases. Second, with a small number of transactions, declared mode outperforms ordered mode by up to 40%. Again, disk traces help to reveal the reason. Ordered mode relies on the sorting provided by the per-file dirty page trees, and therefore its write requests are scattered across the disk. In declared mode, however, the sort performed during commit has a global view of all data being written for the transaction, thus sending the write requests to the device layer in a more efficient order.

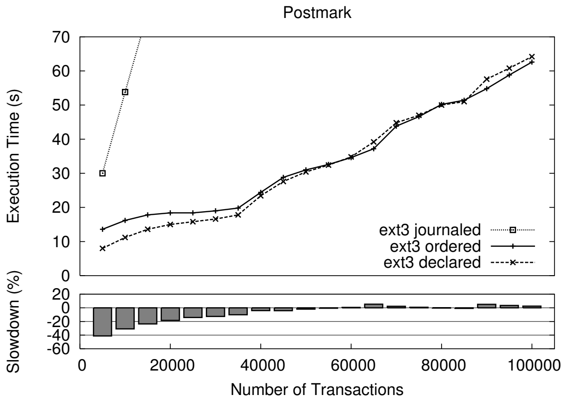


Figure 8: **Postmark Performance.** *The top graph plots the execution time of the postmark benchmark as the number of transactions increases along the x-axis. The bottom graph shows the slowdown of declared mode when compared to ordered mode.*

Finally, we examine the performance of a TPC-B-like workload that performs a financial transaction across

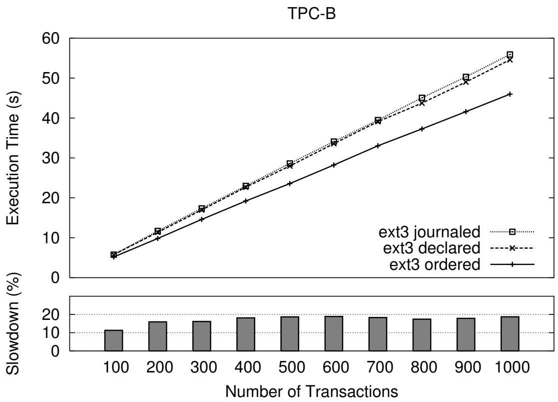


Figure 9: **TPC-B Performance.** *The top graph plots the execution time of the TPC-B benchmark as the number of transactions increases along the x-axis. The bottom graph shows the slowdown of declared mode as compared to ordered mode.*

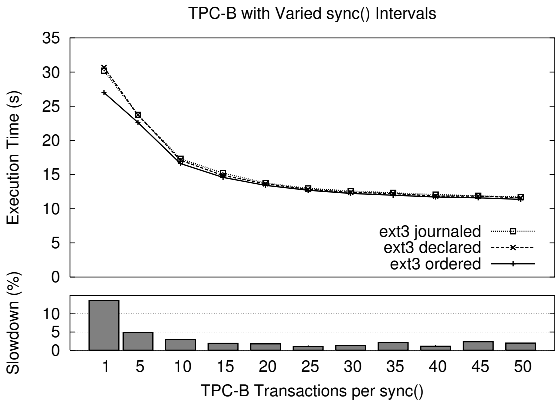


Figure 10: **TPC-B with Varied sync() Intervals.** *The top graph plots the execution time of the TPC-B benchmark as the interval between calls to sync() increases along the x-axis. The bottom graph shows the slowdown of declared mode as compared to ordered mode.*

three files, adds a history record to a fourth file, and commits the changes to disk by calling sync(). The execution time of the benchmark is plotted in Figure 9 as the number of transactions is increased along the x-axis. In this case, declared mode consistently underperforms ext3 ordered mode by approximately 19%, and data-journaling mode performs slightly worse.

The highly synchronous nature of this benchmark presents a worst case scenario for declared mode. Each TPC-B transaction results in a very small ext3 transaction containing only four data blocks, a descriptor block, a journaled metadata block, and a commit block. The declare block at the beginning of each transaction adds 14% overhead in the number of writes performed during the benchmark. To compound this problem, the four data writes are likely serviced in parallel by the array of disks, accentuating the penalty for the declare blocks.

To examine this problem further, we test a modified version of the benchmark that forces data to disk less frequently. This has the effect of increasing the size of each application level transaction, or alternatively simulating concurrent transactions to independent data sets. Figure 10 shows the results of running the TPC-B benchmark with 500 transactions as the interval between calls to sync() increases along the x-axis. As the interval increases, the performance of declared mode and data-journaling mode quickly converge to that of ordered mode. Declared mode performs within 5% of ordered mode for sync() intervals of five or more transactions.

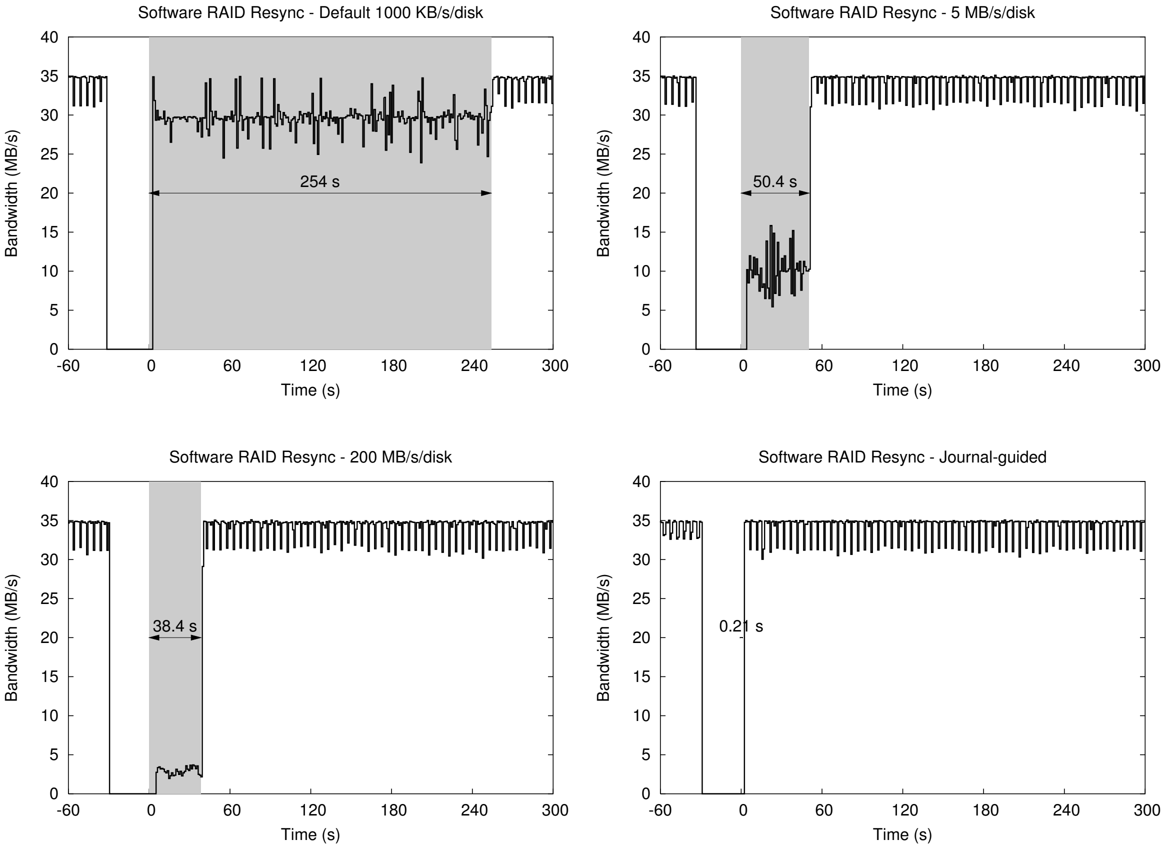
In conclusion, we find that declared mode routinely outperforms data-journaling mode. Its performance is quite close to that of ordered mode, within 5% (and sometimes better) for our random write, sequential write, and file creation microbenchmarks. It also performs within 5% of ordered mode for two macrobenchmarks, ssh and postmark. The worst performance for declared mode occurs under TPC-B with small application-level transactions, but it improves greatly as the effective transaction size increases. Overall, these results indicate that declared mode is an attractive option for enabling journal-guided resynchronization.

5.2 Journal-guided Resynchronization

In our final set of experiments, we examine the effect of journal-guided resynchronization. We expect a significant reduction in resync time, thus shortening the window of vulnerability and improving reliability. In addition, faster resynchronization should increase the amount of bandwidth available to foreground applications after a crash, thus improving their availability. We compare journal-guided resynchronization to the Linux software RAID resync at the default rate and at two other rates along the availability versus reliability spectrum.

The experimental workload consists of a single foreground process performing sequential reads to a set of large files. The amount of read bandwidth it achieves is measured over one second intervals. Approximately 30 seconds into the experiment, the machine is crashed and rebooted. When the machine restarts, the RAID resynchronization process begins, and the foreground process reactivates as well.

Figure 11 shows a series of such experiments plotting the foreground bandwidth on the y-axis as time progresses on the x-axis. Note that the origin for the x-axis coincides with the beginning of resynchronization, and the duration of the process is shaded in grey. The top left graph in the figure shows the results for the default Linux resync limit of 1000 KB/s per disk, which prefers availability over reliability. The process takes 254 seconds



Resync Type	Resync Rate Limit	Foreground Bandwidth	Vulnerability Window	Vulnerability vs. Default
Default	1000 KB/s/disk	29.58 ± 1.69 MB/s	254.00 s	100.00%
Medium	5 MB/s/disk	29.70 ± 9.48 MB/s	50.41 s	19.84%
High	200 MB/s/disk	29.87 ± 10.65 MB/s	38.44 s	15.13%
Journal-guided		34.09 ± 1.51 MB/s	0.21 s	0.08%

Figure 11: **Software RAID Resynchronization.** *The graphs plot the bandwidth achieved by a foreground process performing sequential scans of files on a software RAID array during a system crash and the ensuing array resynchronization. The recovery period is highlighted in grey and its duration is listed. In the first three graphs, the bandwidth allocated to resynchronization is varied: the default of 1000 KB/s per disk, 5 MB/s per disk, and 200 MB/s per disk. The final graph depicts recovery using journal guidance. The table lists the availability of the foreground service and the vulnerability of the array compared to the default resynchronization period of 254 seconds following restart.*

to scan the 1.25 GB of raw disk space in our RAID-5 array. During that time period, the foreground process bandwidth drops to 29 MB/s from the unimpeded rate of 34 MB/s. After resynchronization completes, the foreground process receives the full bandwidth of the array.

Linux allows the resynchronization rate to be adjusted via a sysctl variable. The top right graph in Figure 11 shows the effect of raising the resync limit to 5 MB/s per disk, representing a middle ground between reliability and availability. In this case, resync takes only 50.41 seconds, but the bandwidth afforded the foreground activity drops to only 9.3 MB/s. In the bottom left graph, the resync rate is set to 200 MB/s per disk, favoring reliability over availability. This has the effect of reducing the resync time to 38.44 seconds, but the foreground bandwidth drops to just 2.6 MB/s during that period.

The bottom right graph in the figure demonstrates the use of journal-guided resynchronization. Because of its knowledge of write activity before the crash, it performs much less work to correct any array inconsistencies. The process finishes in just 0.21 seconds, greatly reducing the window of vulnerability present with the previous approach. When the foreground service activates, it has immediate access to the full bandwidth of the array, increasing its availability.

The results of the experiments are summarized in the table in Figure 11. Each metric is calculated over the 254 second period following the restart of the machine in order to compare to the default Linux resynchronization. The 5 MB/s and 200 MB/s resync processes sacrifice availability (as seen in the foreground bandwidth variability) to improve the reliability of the array, reducing the vulnerability windows to 19.84% and 15.13% of the default, respectively. The journal-guided resync process, on the other hand, improves both the availability of the foreground process and the reliability of the array, reducing its vulnerability to just 0.08% of the default case.

It is important to note here that the execution time of the scan-based approach scales linearly with the raw size of the array. Journal-guided resynchronization, on the other hand, is dependent only on the size of the journal, and therefore we expect it to complete in a matter of seconds even for very large arrays.

5.3 Complexity

Table 2 lists the lines of code, counted by the number of semicolons and braces, that were modified or added to the Linux software RAID, ext3 file system, and journaling modules. Very few modifications were needed to add the verify read interface to the software RAID module because the core functionality already existed and merely needed to be activated for the requested stripe. The ext3 changes involved hiding dirty buffers for declared mode and using verify reads during recovery. The majority of the changes occurred in the journaling module for writing declare blocks in the commit phase and performing careful resynchronization during recovery.

Module	**Orig. Lines**	**Mod. Lines**	**New Lines**	**Percent Change**
Software RAID	3475	2	16	0.52%
ext3	8621	22	47	0.80%
Journaling	3472	43	265	8.87%
Total	15568	67	328	2.53%

Table 2: **Complexity of Linux Modifications.** *The table lists the lines of code (counting semicolons and braces) in the original Linux 2.6.11 source and the number that were modified or added to each of the software RAID, ext3 file system, and journaling modules.*

As a point of comparison, the experimental version of Linux RAID-1 bitmap logging consists of approximately 1200 lines of code, a 38% increase over RAID-1 alone. Most of our changes are to the journaling module, increasing its size by about 9%. Overall, our modifications consist of 395 lines of code, a 2.5% change across the three modules. These observations support our claim that leveraging functionality across cooperating layers can reduce the complexity of the software system.

6 Related Work

Brown and Patterson [2] examine three different software RAID systems in their work on availability benchmarks. They find that the Linux, Solaris, and Windows implementations offer differing policies during reconstruction, the process of regenerating data and parity after a disk failure. Solaris and Windows both favor reliability, while the Linux policy favors availability. Unlike our work, the authors do not focus on improving the reconstruction processes, but instead on identifying their characteristics via a general benchmarking framework.

Stodolsky *et al.* [13] examine parity logging in the RAID layer to improve the performance of small writes. Instead of writing new parity blocks directly to disk, they store a log of parity update images which are batched and written to disk in one large sequential access. Similar to our discussion of NVRAM logging, the authors require the use of a fault tolerant buffer to store their parity update log, both for reliability and performance. These efforts to avoid small random writes support our argument that maintaining performance with RAID level logging is a complex undertaking.

The Veritas Volume Manager [19] provides two facilities to address faster resynchronization. A dirty region log can be used to speed RAID-1 resynchronization by examining only those regions that were active before a crash. Because the log requires extra writes, however, the author warns that coarse-grained regions may be needed to maintain acceptable write performance. The Volume Manager also supports RAID-5 logging, but non-volatile memory or a solid state disk is recommended to support the extra log writes. In contrast, our declared mode offers fine-grained journal-guided resynchronization with little performance degradation and without the need for additional hardware.

Schindler *et al.* [11] augment the RAID interface to provide information about individual disks. Their Atropos volume manager exposes disk boundary and track information to provide efficient semi-sequential access to two-dimensional data structures such as database tables. Similarly, E×RAID [4] provides disk boundary and performance information to augment the functionality of an informed file system. Our verify read interface is much less complex, providing file system access to functionality that already exists in the software RAID layer.

7 Conclusions

We have examined the ability of a journaling file system to provide support for faster software RAID resynchronization. In order to obtain a record of the outstanding writes at the time of a crash, we introduce ext3 declared mode. This new mode guarantees to declare its intentions in the journal before writing data to disk. Despite this extra write activity, declared mode performs within 5% of its predecessor.

In order to communicate this information to the software RAID layer, the file system utilizes a new verify read request. This request instructs the RAID layer to read the block and repair its redundant information, if necessary. Combining these features allows us to implement fast, journal-guided resynchronization. This process improves both software RAID reliability and availability by hastening the recovery process after a crash.

Our general approach advocates a system-level view for developing the storage stack. Using the file system journal to improve the RAID system leverages existing functionality, maintains performance, and avoids duplicating complexity at multiple layers. Each of these layers may implement its own abstractions, protocols, mechanisms, and policies, but it is often their interactions that define the properties of a system.

8 Acknowledgements

We would like to thank John Bent, Nathan Burnett, and the anonymous reviewers for their excellent feedback. This work is sponsored by NSF CCR-0092840, CCR-0133456, NGS-0103670, ITR-0325267, Network Appliance, and EMC.

References

[1] S. Best. JFS Overview. www.ibm.com/developerworks/library/l-jfs.html, 2004.

[2] A. Brown and D. A. Patterson. Towards Maintainability, Availability, and Growth Benchmarks: A Case Study of Software RAID Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 263–276, San Diego, California, June 2000.

[3] P. Clements and J. Bottomley. High Availability Data Replication. In *Proceedings of the 2003 Linux Symposium*, Ottawa, ON, Canada, June 2003.

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

[5] EMC. EMC Centera: Content Addressed Storage System. http://www.emc.com/, 2004.

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

[7] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. Fsck - The UNIX File System Check Program. Unix System Manager's Manual - 4.3 BSD Virtual VAX-11 Version, April 1986.

[8] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Chicago, Illinois, June 1988.

[9] H. Reiser. ReiserFS. www.namesys.com, 2004.

[10] 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.

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

[12] D. A. Solomon. *Inside Windows NT (Microsoft Programming Series)*. Microsoft Press, 1998.

[13] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, pages 64–75, San Diego, California, May 1993.

[14] Sun. Solaris Volume Manager Administration Guide. http://docs.sun.com/app/docs/doc/816-4520, July 2005.

[15] D. Teigland and H. Mauelshagen. Volume Managers in Linux. In *Proceedings of the USENIX Annual Technical Conference (FREENIX Track)*, Boston, Massachusetts, June 2001.

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

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

[18] S. C. Tweedie. EXT3, Journaling File System. olstrans.sourceforge.net/ release/OLS2000-ext3/OLS2000-ext3.html, July 2000.

[19] Veritas. Features of VERITAS Volume Manager for Unix and VERITAS File System. http://www.veritas.com/us/products/volumemanager/whitepaper-02.html, July 2005.