Generalized File System Dependencies

Christopher Frost[*•] Mike Mammarella[*•] Eddie Kohler[*]
Andrew de los Reyes[†] Shant Hovsepian[*] Andrew Matsuoka[‡] Lei Zhang[†]
[*]UCLA [†]Google [‡]UT Austin
http://featherstitch.cs.ucla.edu/

ABSTRACT

Reliable storage systems depend at least in part on a "write-before" relationship, where one change to stable storage is delayed until another change commits. A journaled file system, for example, must write and commit each journal transaction before applying that transaction's changes, and soft updates [9] and other consistency enforcement mechanisms have similar constraints, implemented in each case in system-dependent ways. We present a general *patch* abstraction that makes the write-before relationship explicit and file system agnostic. A patch-based file system implementation expresses dependencies among writes to stable storage, but does not enforce specific block write orders to satisfy those dependencies. Storage system modules can examine, preserve, and modify the dependency structure; the buffer cache writes blocks as constrained by that structure. These generalized file system dependencies are naturally exportable to user level. Our patch-based storage system, *Featherstitch*, includes a number of important optimizations. A Featherstitch ext2 prototype running in the Linux kernel supports asynchronous writes, soft updates-like dependencies, and journaling. It performs competitively with ext2 and ext3 on many benchmarks; supports unusual configurations, such as correct dependency enforcement within a loopback file system; and allows applications to specify consistency requirements without micromanaging how those requirements are satisfied.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management; D.4.5 [**Operating Systems**]: Reliability—*Fault-tolerance*; D.4.7 [**Operating Systems**]: Organization and Design

General Terms: Design, Performance, Reliability

Keywords: dependencies, journaling, file systems, soft updates

1 INTRODUCTION

This paper builds a complete and relatively efficient file system layer, called *Featherstitch*, that uses a simple, uniform, and file system agnostic data type to explicitly represent all write-before relationships among changes to stable storage. Write-before relationships, which require some changes to be committed before others, are part of the abstract basis of every mechanism for ensuring file system consistency and reliability, from journaling to synchronous writes. Featherstitch API design and optimizations make these relationships, in a concrete form called *patches*, a promising implementation strategy as well.

A patch represents both a change to disk data and any *dependencies* between that change and other changes. Patches were initially inspired by the dependency abstraction from BSD's soft updates [9], but whereas soft updates dependencies implement a particular type of consistency and involve many structures specific to the UFS file system [17], patches are fully general, specifying only how a range of bytes should be changed. This lets file system implementations specify a write-before relationship between changes without dictating a write order that honors that relationship. It lets storage system components examine and modify dependency structures independent of the file system's layout, possibly even changing one type of consistency into another. It also lets applications modify dependency structures: a user-level interface called *patchgroups* shows that applications can access patches and define consistency policies for the underlying storage system to follow.

A uniform and pervasive patch abstraction may simplify the implementation of, extension of, and experimentation with mechanisms for ensuring file system consistency and reliability. File system implementers find it difficult to provide consistency guarantees [17, 32] and implementations are often buggy [37, 38], a situation further complicated by file system extensions and special disk interfaces [5, 19, 22, 27, 29, 31, 35]. File system extension techniques such as stackable file systems [11, 23, 39, 40] leave consistency up to the underlying file system; any extension-specific ordering requirements are difficult to express at the VFS layer. Although maintaining file system correctness in the presence of failures is increasingly a focus of research [7, 28], other proposed systems for improving file system integrity differ mainly in the kind of consistency they aim to impose, ranging from metadata consistency to full data journaling and full ACID transactions [8, 15]. Some users, however, provide end-to-end reliability for some data and prefer to avoid *any* associated consistency slowdowns in the file system layer [34]. Patches can represent all these choices, and since they provide a common language for file systems and extensions to discuss consistency requirements, they can even allow combinations of consistency protocols to comfortably coexist.

Applications likewise have few mechanisms for controlling buffer cache behavior in today's systems, and robust applications, including databases, mail servers, and source code management tools, must choose between several mediocre options. They can accept the performance penalty of expensive system calls like `fsync` and `sync`, use tedious and fragile sequences of operations that count on particular file system semantics, or rely on specialized raw-disk interfaces. Patchgroups give applications some of the same benefits that patches provide for kernel-level file system implementations and extensions. Modifying an IMAP mail server to use patchgroups required only localized changes. The result meets IMAP's consistency requirements no matter what type of consistency the underlying file system provides, and avoids the perfor-

•Contact authors.
This work was completed while all authors were at UCLA.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.

mance hit of full synchronization.

Though production file systems use system-specific optimizations to achieve consistency without sacrificing performance, we tried to achieve good performance in a general way. A naive patch-based implementation scaled terribly, spending far more space and time on dependency manipulation than conventional systems. We therefore developed several optimizations that attack patch memory and CPU overheads, reducing some overheads by at least at least 99.9% in our tests. Room for improvement remains, but Featherstitch is competitive with equivalent Linux configurations on many benchmarks.

Our contributions include the patch model and design, our optimizations for making patches more efficient, the patchgroup mechanism that exports patches to applications, and several individual Featherstitch modules, such as the journal.

We describe patches abstractly, state their behavior and safety properties, give examples of their use, and reason about the correctness of our optimizations. We then describe the Featherstitch implementation, which is decomposed entirely into pluggable modules that manipulate patches, hopefully making the system as a whole more configurable, extensible, and easier to understand. Following an evaluation comparing Featherstitch and Linux-native file system implementations, we conclude.

2 Related Work

Most modern file systems protect file system integrity in the face of possible power failure or crashes via journaling, which groups operations into transactions that commit atomically [25]. The content and the layout of the journal vary in each implementation, but in all cases, the system can use the journal to replay (or roll back) any transactions that did not complete due to the shutdown. A recovery procedure, if correct [37], avoids time-consuming *fsck*-like file system checks on reboot after a crash in favor of simple journal operations.

Soft updates [9] is another important mechanism for ensuring post-crash consistency. Carefully managed write orderings avoid the need for synchronous writes to disk or duplicate writes to a journal; only relatively harmless inconsistencies, such as leaked blocks, are allowed to appear on the file system. As in journaling, soft updates can avoid *fsck* after a crash, although a background *fsck* is required to recover leaked storage.

Both journaling and soft updates are naturally represented in terms of patches, as Section 3.4 demonstrates, and we use both journaling and soft updates as running examples throughout the paper. In both cases, the patch implementation extracts ad hoc orderings and optimizations into general dependency graphs, making the orderings potentially easier to understand and modify. Soft updates is in some ways a more challenging test of the patch abstraction: its dependencies are more variable and harder to predict, it is widely considered difficult to implement, and the existing FreeBSD implementation is quite optimized [17]. We therefore frequently discuss soft updates-like dependencies. This should not be construed as a wholesale endorsement of soft updates, which relies on a property (atomic block writes) that many disks do not provide, and which often requires more seeks than journaling despite writing less data.

CAPFS [33] and Echo [16] considered customizable application-level consistency protocols in the context of distributed, parallel file systems. CAPFS allows application writers to design plug-ins for a parallel file store that define what actions to take before and after each client-side system call. These plug-ins can enforce additional consistency policies. Echo maintains a partial order on the locally cached updates to the remote file system, and guarantees that the server will store the updates accordingly; applications can extend the partial order. Both systems are based on the principle that not providing the right consistency protocol can cause unpredictable failures, yet enforcing unnecessary consistency protocols can be extremely expensive. Featherstitch patchgroups generalize this sort of customizable consistency and brings it to disk-based file systems.

A similar application interface to patchgroups is explored in Section 4 of Burnett's thesis [4]. However, the methods used to implement the interfaces are very different: Burnett's system tracks dependencies at the level of entire system calls, associates dirty blocks with unique IDs returned by those calls, and duplicates dirty blocks when necessary to preserve ordering. Featherstitch tracks individual changes to blocks internally, allowing kernel modules a finer level of control, and only chooses to expose a userspace interface similar to Burnett's as a means to simplify the sanity checking required of arbitrary user-submitted requests. Additionally, our evaluation uses a real disk rather than trace-driven simulations.

Dependencies have been used in BlueFS [20] and xsyncfs [21] to reduce the aggregate performance impact of strong consistency guarantees. In particular, xsyncfs provides *external synchrony*, which has the same consistency guarantees as synchronous file system ordering for users, but does not actually block applications on each write. Writes are committed in groups using a journaling design, but xsyncfs enforces additional write-before relationships on *non-file system* communication: a journal transaction must commit before output from any process involved in that transaction can escape the kernel, for example via the terminal or a network connection. These dependency relationships are tracked across IPC as well. Featherstitch patches could generalize the link between the file system implementation and xsyncfs dependencies, or the cross-network dependencies sent in BlueFS; for instance, Featherstitch patches apply at least conceptually to any file system, whereas the xsyncfs implementation assumes a file system like ext3. Conversely, Featherstitch applications could benefit from the combination of strict ordering and nonblocking writes provided by xsyncfs.

Stackable module software for file systems continues to attract active research [10, 11, 23, 30, 35, 36, 39, 40], as do other extensions to file system and disk interfaces [12, 26]; these systems could potentially benefit from a patch-like mechanism that represented write-before relationships and consistency requirements in an agnostic manner.

Featherstitch adds to this body of work by designing a primitive that generalizes and makes explicit the write-before relationship present in many storage systems, and implementing a storage system in which that primitive is pervasive throughout.

3 Patch Model

Every change to stable storage in a Featherstitch system is represented by a *patch*. This section describes the basic patch abstraction and describes our implementation of that abstraction.

3.1 Disk Behavior

We first describe how disks behave in our model, and especially how disks commit patches to stable storage. Although our terminology originates in conventional disk-based file systems with uniformly-sized blocks, the model would apply with small changes to file systems with non-uniform blocks and to other media, including RAID and network storage.

We assume data is committed to stable storage in units called **blocks**. All writes affect one or more blocks, and it is impossible to selectively write part of a block.

A **patch** models any change to block data. Each patch applies to exactly one block, so a file system change that affects n blocks

p	a patch
$B[p]$	patch p's block
$\mathcal{C}, \mathcal{U}, \mathcal{F}$	the sets of all committed, uncommitted, and in-flight patches, respectively
$\mathcal{C}_B, \mathcal{U}_B, \mathcal{F}_B$	committed/uncommitted/in-flight patches on block B
$q \rightsquigarrow p$	q depends on p (p must be written before q)
$dep[p]$	p's dependencies: $\{x \mid p \rightsquigarrow x\}$
$q \rightarrow p$	q directly depends on p ($q \rightsquigarrow p$ means either $q \rightarrow p$ or $\exists x : q \rightsquigarrow x \rightarrow p$.)
$dep_1[p]$	p's direct dependencies: $\{x \mid p \rightarrow x\}$

Figure 1: Patch notation.

requires at least n patches to represent. Each patch is either **committed**, meaning written to disk; **uncommitted**, meaning not written to disk; or **in flight**, meaning in the process of being written to disk. The intermediate in-flight state models caching, reordering, and delay in the storage layer: blocks written by an operating system may be committed in any order, possibly after disk scheduling and a period in the on-disk cache. Patches are created as uncommitted. The operating system moves uncommitted patches to the in-flight state by writing their blocks to the disk controller. Some time later, the disk writes these blocks to stable storage and reports success; when the processor receives this acknowledgment, it commits the relevant patches. Committed patches stay committed permanently, although their effects can be undone by subsequent patches. The sets $\mathcal{C}$, $\mathcal{U}$, and $\mathcal{F}$ represent all committed, uncommitted, and in-flight patches, respectively.

Patch p's block is written $B[p]$. Given a block B, we write $\mathcal{C}_B$ for the set of committed patches with block B, or in notation $\mathcal{C}_B = \{p \in \mathcal{C} \mid B[p] = B\}$. $\mathcal{F}_B$ and $\mathcal{U}_B$ are defined similarly.

We can now model disk controllers' behavior: they write in-flight patches one block at a time, as follows.

1. Pick some block B with $\mathcal{F}_B \neq \varnothing$.
2. Commit and acknowledge each patch in $\mathcal{F}_B$.

Disks perform better when allowed to reorder requests, so operating systems try to keep many blocks in flight. Although file systems will generally write all of a block's uncommitted patches when they write the block, they may, if they choose, write a *subset* of those patches—and, as we will see, this is sometimes required to preserve write-before relationships.

We intentionally do not specify whether the storage system writes blocks atomically. Some file system designs, such as soft updates, rely on block write atomicity: soft updates expects that if the disk fails while a block B is in flight, then B's value on recovery must equal either the old value or the new value. Most journal designs do not require this, and include recovery procedures that handle in-flight block corruption—for instance, because the memory holding the new value of the block lost coherence before the disk stopped writing. Since patches model the write-before relationships used to build these journal designs, patches do not provide block atomicity themselves, and a patch-based file system with soft updates-like dependencies should only be used in conjunction with a storage layer providing block atomicity. (This is no different from other soft updates implementations.)

3.2 Dependencies

A patch-based storage system implementation represents write-before relationships using an explicit **dependency** relation. The disk controller and lower layers don't understand dependencies; instead, the system maintains dependencies and passes blocks to the controller in an order that preserves dependency semantics. Patch q *depends on* patch p, written $q \rightsquigarrow p$, when the storage system must commit p before q. (As a special case, if p and q are on the same block, they may be committed at the same time.) The file system creates dependencies that express its desired consistency semantics. For example, a file system with asynchronous writes (and no durability guarantees) might create patches with no dependencies at all; a file system wishing to strictly order writes might set $p_n \rightsquigarrow p_{n-1} \rightsquigarrow \cdots \rightsquigarrow p_1$. Circular dependencies among patches cannot be resolved and are therefore errors; for example, neither p nor q could be written before the other if $p \rightsquigarrow q \rightsquigarrow p$. (Although a circular dependency chain entirely within a single block avoids this problem, Featherstitch treats all circular chains as errors.) Patch p's *set* of dependencies, written $dep[p]$, consists of all patches on which p depends; $dep[p] = \{x \mid p \rightsquigarrow x\}$. Given a set of patches P, we write $dep[P]$ to mean the combined dependency set $\bigcup_{p \in P} dep[p]$.

The **disk safety property** formalizes dependency requirements by stating that the dependencies of all committed patches must have been committed:

$$dep[\mathcal{C}] \subseteq \mathcal{C}.$$

Thus, no matter when the system crashes, the disk is consistent in terms of dependencies. Since, as described above, the disk controller can write blocks in any order, a Featherstitch storage system must also ensure that in-flight blocks are independent. This is precisely stated by the **in-flight safety property:**

$$\text{For any block } B,\ dep[\mathcal{F}_B] \subseteq \mathcal{C} \cup \mathcal{F}_B.$$

Given this, $dep[\mathcal{F}_B] \cap dep[\mathcal{F}_{B'}] \subseteq \mathcal{C}$ for any $B' \neq B$, and the disk controller can write in-flight blocks in any order and still preserve disk safety. Finally, to uphold the in-flight safety property, the buffer cache must write blocks as follows:

Pick some block B with $\mathcal{U}_B \neq \varnothing$ and $\mathcal{F}_B = \varnothing$.
Pick some $P \subseteq \mathcal{U}_B$ with $dep[P] \subseteq P \cup \mathcal{C}$.
Move each $p \in P$ to $\mathcal{F}$ (in-flight).

We require that $\mathcal{F}_B = \varnothing$ to ensure that at most one version of a block is in flight at any time.

In summary, the main Featherstitch implementation challenge is to design data structures that make it easy to create patches, quick to manipulate patches, and that help the buffer cache write blocks and patches in line with the above procedure. (The buffer cache is also expected to write all blocks eventually, a liveness property.)

3.3 Dependency Implementation

The write-before relationship is transitive, so if $r \rightsquigarrow q$ and $q \rightsquigarrow p$, there is no need to explicitly store an $r \rightsquigarrow p$ dependency. To reduce storage requirements, a Featherstitch implementation maintains only a subset of dependencies called the *direct dependencies*. Each patch p has a corresponding set of direct dependencies $dep_1[p]$; we say q *directly depends on* p, and write $q \rightarrow p$, when $p \in dep_1[q]$. The dependency relation $q \rightsquigarrow p$ simply means that either $q \rightarrow p$ or $q \rightsquigarrow x \rightarrow p$ for some patch x.

Another implementation choice defines how the buffer cache can write a subset of a block's patches. We maintain each block in its dirty state, including the effects of all uncommitted patches, but each patch carries **undo data**: the previous version of the block data altered by the patch. If a patch p is not written with its containing block, the buffer cache *reverts* the patch, which swaps the new data on the buffered block and the previous version in the undo data. Once the block is written, the system will re-apply the patch and, when allowed, write the block again, this time including the patch. Undo information adds greatly to memory and CPU utilization, but it can often be optimized away, as we show in Section 4.

Figure 1 summarizes our patch notation.

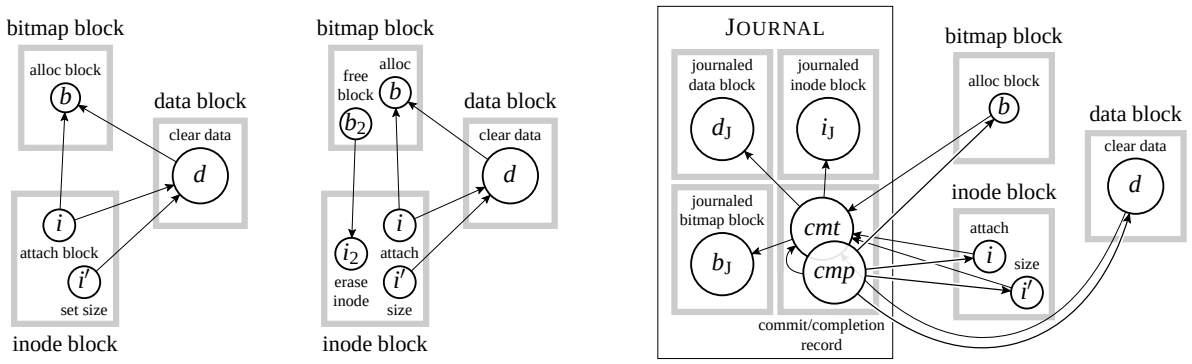


Figure 2: Example patch arrangements for an ext2-like file system. Circles represent patches, shaded boxes represent disk blocks, and arrows represent dependencies. **a)** A soft updates order for appending a zeroed-out block to a file. **b)** A different file on the same inode block is removed before the previous changes commit, inducing a circular block dependency. **c)** A journal order for appending a zeroed-out block to a file.

3.4 Examples

This section illustrates how patches can implement two widely-used file system consistency mechanisms, soft updates and journaling. Our basic example extends an existing file by a single block—perhaps an application calls `ftruncate` to append 1024 zero bytes to an empty file. The file system is based on ext2, an FFS-like file system with inodes and a free block bitmap. In such a file system, this operation requires (1) allocating a block by marking the corresponding bit as "allocated" in the free block bitmap, (2) attaching the block to the file's inode, (3) setting the inode's size, and (4) clearing the allocated data block. These operations affect three blocks—a free block bitmap block, an inode block, and a data block—and correspond to four patches: b (allocate), i (attach), i' (size), and d (clear).

Soft updates Early file systems aimed to avoid post-crash disk inconsistencies by writing some, or all, blocks synchronously. For example, the write system call might block until all metadata writes have completed—clearly bad for performance. Soft updates, in contrast, provides post-crash consistency without synchronous writes by tracking and obeying necessary dependencies among writes [9]. A soft updates file system orders its writes to enforce three simple rules for metadata consistency [9]:

1. "Never write a pointer to a structure until it has been initialized (e.g., an inode must be initialized before a directory entry references it)."
2. "Never reuse a resource before nullifying all previous pointers to it".
3. "Never reset the last pointer to a live resource before a new pointer has been set".

By following these rules, a file system limits the possible inconsistencies present on the disk at any time to leaked resources, such as blocks or inodes marked as in use but unreferenced. The file system can be used immediately on reboot; a background scan can locate and recover the leaked resources while the system is in use.

These rules map directly to Featherstitch. Figure 2a shows a set of soft updates-like patches and dependencies for our block-append operation. Rule 1 requires that $i \to b$. Rule 2 requires that d depend on previous pointers to the block being nullified; a simple, though more restrictive, way to accomplish this is to let $d \to b$, where b depends on any such nullifications (there are none here). The dependencies $i \to d$ and $i' \to d$ provide an additional guarantee above and beyond metadata consistency, namely that no file ever contains accessible uninitialized data. A similar dependency would make inode updates depend on actual data writes. In contrast, the BSD UFS soft updates implementation represents each UFS operation by a different specialized structure encapsulating all of that operation's disk changes and dependencies. These structures, their relationships, and their uses are quite complex [17].

Figure 2b shows how an additional file system operation can induce a circular dependency among blocks. Before the changes in Figure 2a are written, the user deletes a one-block file whose data block and inode happen to lie on the same bitmap and inode blocks used by the previous operation. Rule 2 requires the dependency $b_2 \to i_2$; but given this dependency and the previous $i \to b$, neither the bitmap block nor the inode block can be written first! A solution is to roll back patch b_2 by undoing its effects. The resulting bitmap block, which contains only b, is safe to write. Once this write commits, all of i, i', and i_2 are safe to write. When these in turn commit, the system can write the bitmap block again, this time including b_2.

Journal transactions A journaling file system ensures post-crash consistency using a write-ahead log. All changes in a transaction are first copied into an on-disk journal. Once these copies commit, a *commit record* is written to the journal, signaling that the transaction is complete and all its changes are valid. Once the commit record is written, the original changes can be written to the file system in any order, since after a crash the system can replay the journal transaction to recover. Finally, once all the changes have been written to the file system, the commit record can be erased, allowing that portion of the journal to be reused.

This process also maps directly to patch dependencies, as shown in Figure 2c. Copies of the affected blocks are written into the journal area using patches d_J, i_J, and b_J, each on its own block. Patch *cmt* creates the commit record on a fourth block in the journal area; it depends on d_J, i_J, and b_J. The changes to the main file system all depend on *cmt*. Finally, patch *cmp*, which depends on the main file system changes, overwrites the commit record with a completion record. Again, a circular block dependency requires the system to roll back a patch, namely *cmp*, and write the commit/completion block twice.

3.5 Implementation and Empty Patches

The Featherstitch file system implementation creates patch structures corresponding directly to our abstraction. Creation functions have names like `patch_create_byte`; their arguments include the relevant block, any direct dependencies, and the new data. Most patches specify this data as a contiguous byte range, including an

offset into the block and the patch length in bytes. The undo data for very small patches (4 bytes or less) is stored in the patch structure itself; for larger patches, undo data is stored in separately allocated memory. In bitmap blocks, changes to individual bits in a word can have independent dependencies, which we handle with a special bit-flip patch type.

The implementation automatically detects one special type of dependency. If two patches p and q affect the same block and have overlapping data ranges, and p was created before q, then Featherstitch adds an *overlap dependency* $q \to p$ to ensure that p is written before q. There's no need for file system modules to detect and enforce such dependencies themselves.

For each block B, Featherstitch maintains a list of all patches with $B[p] = B$. However, it does not track committed patches; when patch p commits, Featherstitch destroys p's data structure and removes all dependencies $q \to p$. The resulting behavior follows our model. For each patch p, Featherstitch maintains lists of its direct dependencies and its direct "reverse dependencies" (that is, all q where $q \to p$).

The implementation also supports *empty* patches, which have no associated data or block. For example, during a journal transaction, changes to the main body of the disk should depend on a journal commit record that has not yet been created. Featherstitch makes these patches depend on an empty patch that is explicitly held in memory. Once the commit record is created, the empty patch is updated to depend on the actual commit record and then released. The empty patch automatically commits at the same time as the commit record, allowing the main file system changes to follow. Empty patches can shrink memory usage by representing quadratic sets of dependencies with a linear number of edges: if all m patches in Q must depend on all n patches in P, one could add an empty patch e and $m+n$ direct dependencies $q_i \to e$ and $e \to p_j$. This is useful for patchgroups as described in §5. However, extensive use of empty patches adds to system time by requiring that functions traverse empty patch layers to find true dependencies. Our implementation therefore uses empty patches infrequently, and in the rest of this paper, patches are nonempty unless explicitly stated.

3.6 Discussion

The patch abstraction seems general enough to suit many forms of dependency tracking. Only one property of the model depends on the conventional file system context, namely that circular dependency chains are errors. This restriction is required because Featherstitch assumes a lower layer that commits one block at a time. Disks certainly behave this way, but a dependency tracker built above a more advanced lower layer—such as a journal—could resolve many circular dependency chains by forcing the relevant blocks into a single transaction or transaction equivalent. Featherstitch's journal module could potentially implement this, allowing upper layers to create (size-limited) circular dependency chains, but we leave the implementation for future work.

Patches model write-before relationships, but one might instead build a generalized file system dependency system that modeled abstract transactions. We chose write-before relationships as our foundation since they minimally constrain file system disk layout; nevertheless, it would be interesting to consider whether a transaction abstraction could model, say, soft updates-like dependencies.

4 Patch Optimizations

Assume Featherstitch is running an ext2 file system with soft updates-like dependencies and 4 kB blocks. Figure 3a shows the patches generated when an application appends 16 kB of data to an existing empty file using four 4 kB writes. This change requires allocating four blocks (patches b_1–b_4), writing data to them (d_1–d_4 and d'_1–d'_4), attaching them to the file's inode (i_1–i_4), changing the inode's file size and modification time (i'_1–i'_4 and i''), and updating the "group descriptor" and superblock to account for the allocated blocks (g and s). All of the one-block writes update the inode; note, for example, how overlap dependencies force each modification of the inode's size to depend on the previous one. A total of eight blocks are written during the operation. Unoptimized Featherstitch, however, represents the operation with 23 patches and roughly 33,000 (!) bytes of undo data. The patches slow down the buffer cache system by making graph traversals more expensive; storing undo data for patches on data blocks is particularly painful here, as data blocks will *never* need to be reverted; and in larger examples, the effects are even worse: when running the benchmarks described in §8, unoptimized Featherstitch allocates twice as much memory for undo data as for cached block data, or even more.

This section shows how generic patch properties and dependency analysis can reduce the 23 patches and 33,000 bytes of undo data in Figure 3a to the 8 patches and 0 bytes of undo data in Figure 3d. Additional optimizations simplify Featherstitch's other main overhead, the CPU time required for the buffer cache to find a suitable set of patches to write. These optimizations apply transparently to any Featherstitch file system, and as we demonstrate in our evaluation, have dramatic effects on real benchmarks as well.

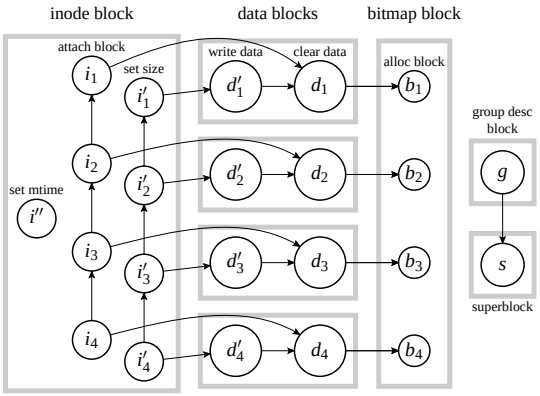
4.1 Hard Patches

The first optimization reduces space overhead by eliminating undo data. When a patch p is created, Featherstitch conservatively detects whether p might require reversion: that is, whether any possible future patches and dependencies could force the buffer cache to undo p before making further progress. If no future patches and dependencies could force p's reversion, then p does not need undo data. The challenge is to detect this condition without predicting the future. We solve this challenge by restricting dependency creation.

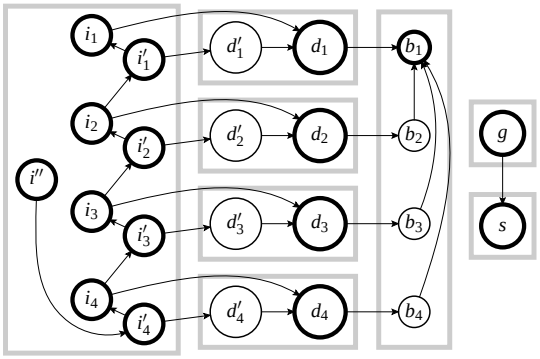
We implemented support for **hard patches**, which are simply patches that lack undo data. Since a hard patch h cannot be reverted, any other patch on its block must depend on h (the other patches can't be written without h). We enforce this requirement, for example using overlap dependencies, and as a result, the buffer cache will write a block's hard patches (if any) whenever it writes the block. The system aims to reduce memory usage by making most patches hard.

But which patches can be made hard? Define a *block-level cycle* as a dependency chain of uncommitted patches $p_n \rightsquigarrow \cdots \rightsquigarrow p_1$ where the ends have the same block $B[p_n] = B[p_1]$, and at least one patch in the middle has a different block $B[p_i] \neq B[p_1]$. The patch p_n is called the *head* of the block-level cycle. Now assume that a patch $p \in \mathcal{U}$ is not the head of any block-level cycle. One can then show that the buffer cache can always write a block without rolling back p. The hard case is where $B[p]$ cannot itself be written without rolling back p, which occurs when p has an uncommitted dependency q on a different block. However, we know that q's uncommitted dependencies, if any, are all on blocks other than p's; otherwise there would be a block-level cycle. Since Featherstitch disallows circular dependencies, every chain of dependencies starting at q has finite length, and therefore contains an uncommitted patch x all of whose dependencies have been committed. (If x has in-flight dependencies, simply wait for the disk controller to commit them.) Since x is not on p's block, the buffer cache can write x without rolling back p.

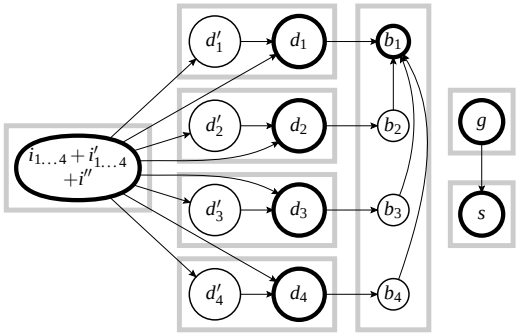
Featherstitch may thus make a patch hard when it can prove that patch will never be the head of a block-level cycle. This requires two tricks. First, an API restriction allows us to search for *existing*,



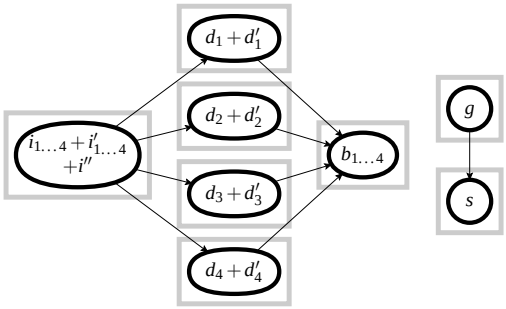
a) Naive implementation



b) With hard patches ...



c) ... plus hard patch merging ...



d) ... plus overlap merging

Figure 3: Patches required to append 4 blocks to an existing file, without and with optimization. Hard patches are shown with heavy borders.

rather than future, block-level cycles: *A patch's direct dependencies are all supplied at creation time*. After p is created, the system can add new dependencies $q \rightarrow p$, but will never add new dependencies $p \rightarrow q$.[1] Since every patch follows this rule, all possible block-level cycles with head p are present in the dependency graph when p is created. Featherstitch must still check for these cycles, of course, and actual graph traversals proved expensive. We thus implemented a conservative approximation: patch p is created as hard if *no* patches on other blocks depend on uncommitted patches on $B[p]$. That is, given any dependency between uncommitted patches $y \rightsquigarrow x$, either $B[x]$ isn't on p's block or *both* x and y are on p's block. If this holds, then p cannot head a block-level cycle no matter its dependencies. This heuristic works well in practice and, given some bookkeeping, takes $O(1)$ time to check.

Applying hard patch rules to our example makes 14 of the 23 patches hard (Figure 3b), reducing the undo data required by slightly more than half.

4.2 Hard Patch Merging

File operations such as block allocations, inode updates, and directory updates create many distinct patches. Keeping track of these patches and their dependencies requires memory and CPU time. Featherstitch therefore *merges* patches when possible, drastically reducing patch counts and memory usage, by conservatively identifying when a new patch could always be written at the same time as an existing patch. Rather than creating a new patch in this case, Featherstitch updates the data and dependencies so as to merge the new patch into the existing one.

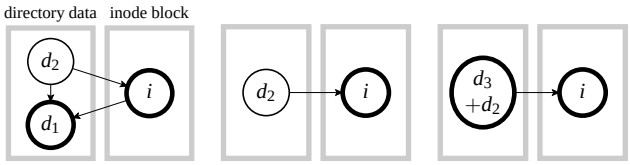
Two types of patch merging involve hard patches, and the first is trivial to explain: since all of a block's hard patches *must* be written at the same time, they can always safely be merged. Featherstitch thus ensures that each block contains at most one hard patch. If Featherstitch detects that a new patch p could be created as hard and p's block already contains a hard patch h, then the implementation merges p into h by applying p's data to the block and setting $dep_1[h] \leftarrow dep_1[h] \cup dep_1[p]$. The existing hard patch h is returned to the caller. This changes h's direct dependency set after h's creation time, but since p could have been created hard, the change cannot introduce any new block-level cycles. Unfortunately, the merge can create *intra*-block cycles. If some empty patch e has $p \rightsquigarrow e \rightsquigarrow h$, then after the merge $h \rightsquigarrow e \rightsquigarrow h$. Featherstitch detects and prunes these cyclic dependencies as p's dependencies are merged into h.

Hard patch merging is able to eliminate 8 of the patches in our running example, as shown in Figure 3c.

Second, Featherstitch detects when a new hard patch can be merged with a block's existing *soft* patches. Block-level cycles may force a patch p to be created as soft. Once those cycles are broken (because the relevant patches commit), p could be converted to hard; but to avoid unnecessary work, Featherstitch delays the conversion, performing it only when it detects that a new patch on p's block could be created hard. Figure 4 demonstrates this scenario using soft updates-like dependencies.

Specifically, consider a new hard patch h added to a block that contains some soft patch p. Since h is considered to overlap p, Featherstitch adds a direct dependency $h \rightarrow p$. Since h could be hard even including this overlap dependency, we know there are no block-level cycles with head h. But as a result, we know that there are no block-level dependency cycles with head p, since any

[1] The actual rule is somewhat more flexible: modules may add new direct dependencies if they guarantee that those dependencies don't produce any new block-level cycles. As one example, if no patch depends on some empty patch e, then adding a new $e \rightarrow q$ dependency can't produce a cycle.



a) Block-level cycle **b)** d_1 commits **c)** After merge

Figure 4: Soft-to-hard patch merging. **a)** Soft updates-like dependencies among directory data and an inode block. d_1 deletes a file whose inode is on i, so Rule 2 requires $i \rightarrow d_1$; d_2 allocates a file whose inode is on i, so Rule 1 requires $d_2 \rightarrow i$. **b)** Writing d_1 removes the cycle. **c)** d_3, which adds a hard link to d_2's file, initiates soft-to-hard merging.

such cycle $p \rightsquigarrow \cdots \rightsquigarrow p_1$ would imply an h-headed cycle $h \rightarrow p \rightsquigarrow \cdots \rightsquigarrow p_1$. Thus, p can be transformed into a hard patch. Featherstitch performs this transformation and merges p and h via hard patch merging.

4.3 Overlap Merging

The final type of merging merges soft patches with other patches, hard or soft, when they overlap. Bitmap blocks, inodes, and directory entries accumulate many nearby and overlapping patches as data is appended to or truncated from a file and as files are created and removed. Figure 3 shows how even data blocks can collect overlapping dependencies: actual data writes d'_j overlap, and therefore depend on, block initialization writes d_j, but cannot be made hard since when they are created another block (the inode) already depends on the data block. Luckily, simple reasoning can identify many mergeable pairs, further reducing the number of patches and the amount of undo data required.

Overlapping patches p_1 and p_2, with $p_2 \rightsquigarrow p_1$, may be merged *unless* some possible future patches and dependencies could force the buffer cache to undo p_2, but not p_1. Here we may reuse the reasoning developed for hard patches above, arriving at a simpler property. If p_2 will never be the head of a block-level cycle containing p_1, then p_2 and p_1 can always be committed together; and any such block-level cycle will exist when p_2 is created.

As with hard patch creation, the Featherstitch implementation checks a simpler property that limits required graph traversals. We experimented with various heuristics to balance CPU expense and missed merge opportunities. The final heuristic examines all dependency chains of uncommitted patches starting at p_2. It succeeds if p_1 is not in any of the chains, failing conservatively if any of the chains grows too long (more than 10 links) or there are too many chains. However, some chains cannot induce block-level cycles and are allowed regardless of length: If $p_1 \rightsquigarrow x$ for some x, then since there are no circular dependencies, any chain from p_2 that encounters x will never encounter p_1 later. Our heuristic detects several easy-to-check instances of this principle, including h, a hard patch on p_1's block ($p_2 \rightsquigarrow h$ by definition). We tuned the heuristic to cope with long dependency chains created by the Andrew benchmark on soft updates-like dependencies. If all chains fit, then there are no block-level cycles from p_2 to p_1, p_2 and p_1 can have the same lifetime, and p_2 can be merged into p_1 where they overlap (which may require growing p_1's data range). It also simplifies the implementation to avoid overlap merging when p_2 overlaps with two or more soft patches that do not themselves overlap.

In our running example, overlap merging is able to combine all remaining soft patches with their hard counterparts, reducing the number of patches to the minimum of 8 and the amount of undo data to the minimum of 0. In our experiments, we observe that hard patches and our patch merging optimizations reduce the amount of memory allocated for undo data in soft updates and journaling orderings by at least 99.9%.

4.4 Ready Patch Lists

Another important optimization greatly reduces CPU time spent in the Featherstitch buffer cache. The buffer cache's main task is to choose sets of patches P that satisfy the in-flight safety property $dep[P] \subseteq P \cup \mathcal{C}$. A naive implementation would simply traverse the dependency graph starting at these patches, looking for problematic dependencies. Patch merging can reduce the size of these traversals by combining patches together. Unfortunately, even modest traversals become painfully slow when executed on every block in a large buffer cache, and in our initial implementation these traversals were a performance bottleneck for even modest cache sizes. Featherstitch therefore precomputes much of the information required for the buffer cache to choose a set of patches to write.

Featherstitch explicitly tracks, for each patch, how many of its direct dependencies remain uncommitted or in flight. These counts are incremented as patches are added to the system, and decremented as the system receives commit notifications from the disk. When both counts reach zero, the patch is safe to write, and it is moved into a *ready list* on its containing block. The buffer cache, then, can immediately tell whether any of a block's patches are writable by examining its ready list.

To write a block B, the buffer cache initially populates the set P with the contents of the ready list. While moving a patch p into P, Featherstitch checks whether there exist dependencies $q \rightarrow p$ where q is also on block B. The system can potentially write q at the same time as p, so q's counts are updated as if p has already committed. This may make q ready, after which it in turn is added to P. (This premature accounting is safe because the system won't try to write B again until p actually commits.)

On-line maintenance of the ready counts adds some cost to several patch manipulations, but since it saves so much duplicate work in the buffer cache the resulting system is more efficient by multiple orders of magnitude—and in particular, CPU time no longer scales superlinearly with the size of the cache.

4.5 Other Optimizations

Optimizations can only do so much with bad sets of dependencies. Just as having too few dependencies can compromise system correctness, having too many dependencies, or the wrong dependencies, can non-trivially degrade system performance. For example, in both the following patch arrangements, s depends on all of r, q, and p, but the left-hand arrangement gives the system more freedom to reorder block writes:

If q, r, and s are adjacent on disk, the left-hand arrangement can be satisfied with two disk requests while the right-hand one will require four. Although the arrangements have similar coding difficulty, in several cases we discovered that one of our file system implementations was performing slowly because it created an arrangement like the one on the right.

Care must be taken to avoid unnecessary *implicit* dependencies, and in particular overlap dependencies. For instance, inode blocks contain multiple inodes, and changes to two inodes should generally be independent; a similar statement holds for directories and even sometimes for different fields in a summary block like the superblock. The best results can be obtained from *minimal* patches that change one independent field at a time. Featherstitch will merge

these patches when appropriate, but if they cannot be merged, minimal patches cause fewer patch revert operations and more flexibility in write ordering.

File system implementations can generate better dependency arrangements when they can detect that certain states will never appear on disk. For example, soft updates requires that clearing an inode depend on nullifications of all the corresponding directory entries, which normally induces dependencies from the inode onto the directory entries. However, if a directory entry will never exist on disk—for example, because a patch to remove the entry merged with the patch that created it—then there is no need to require the corresponding dependency. Leaving out these dependencies can speed up the system by avoiding circular block dependencies, such as those in Figure 4, and thus avoiding the attendant rollbacks and double writes. The Featherstitch ext2 module implements this optimization and another, related one: if a directory entry will never reference an inode, then the patches that clear the corresponding inode bitmap and block bitmap entries need not depend on the patch that deleted that directory entry. These optimizations significantly reduce the number of disk writes, number of patches created, and amount of undo data allocated when files are created and deleted within a short time.

Finally, block allocation policies can have a dramatic effect on the number of I/O requests required to write changes to the disk. For instance, soft updates-like dependencies require that data blocks be initialized before an indirect block references them. dependencies implementing soft updates, indirect block data cannot be written until the referenced blocks have been at least initialized. By allocating an indirect block in the middle of a range of data blocks for a file, the data blocks must be written as two smaller I/O requests since the indirect block cannot be written at the same time. Allocating the indirect block somewhere else allows the data blocks to be written in one larger I/O request, at the cost of (depending on readahead policies) a potential slowdown in read performance.

5 PATCHGROUPS

This section describes *patchgroups*, a simple API that extends patches to user space and lets applications express dependencies among file system operations. Currently, robust applications can ensure their on-disk data remains in a sane state even after a system crash by enforcing write-before relationships in one of two ways: either forcing an ordering by requiring immediate syncs (e.g. `sync`, `fsync`, or `sync_file_range`), or assuming an ordering by depending on particular file system implementation semantics (e.g. journaling). The patch abstraction, in contrast, can let a process specify *what* ordering is required and leave the storage system to implement that ordering in the most appropriate way. Compared to forcing block writes in the application, this lets the file system buffer, combine, and reorder block writes, but is still relatively independent of the underlying file system. For example, patchgroups even allow an application to specify dependencies among raw block device writes.

In this section we describe the patchgroup abstraction and apply it to three robust applications.

5.1 Interface and Implementation

Patchgroups encapsulate sets of file system operations into units among which dependencies can be applied. The patchgroup interface is as follows:

```
typedef int pg_t;           pg_t pg_create(void);
int pg_engage(pg_t pg);     int  pg_disengage(pg_t pg);
int pg_sync(pg_t pg);       int  pg_close(pg_t pg);
int pg_depend(pg_t after, pg_t before);
```

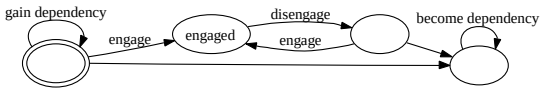


Figure 5: Valid patchgroup dependency operation state machine.

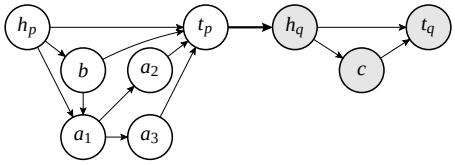


Figure 6: Patches corresponding to two patchgroups, p and q. The h and t patches are created by the patchgroup module; the heavy dependency between t_p and h_q was added by `pg depend(p, q)`. Each of a_i, b, and c corresponds to a different file system change.

Each process has its own set of patchgroups, which are currently shared among all threads.. The call `pg_depend(q, p)` makes patchgroup `q` depend on patchgroup `p`: all patches associated with `p` will be committed prior to those associated with `q`. The patches associated with a patchgroup are those created while the patchgroup is engaged. `pg_sync` forces an immediate write of a patchgroup to disk. `pg_create` creates a new patchgroup and returns its ID and `pg_close` disassociates a patchgroup ID from the underlying patches which implement it.

Whereas Featherstitch modules are presumed to not create cyclic dependencies, the kernel cannot safely trust user applications to be so well behaved, so the patchgroup API restricts dependency manipulation so that cycles are inconstructible. Figure 5 shows when a given patchgroup dependency operation is valid. For example, `pg_depend(q, p)` returns an error if `q` has *ever* been engaged; if it succeeds, a subsequent `pg_engage(p)` will return an error.[2]

Patchgroups and file descriptors are managed similarly—they are copied across `fork`, preserved across `exec`, and closed on `exit`. This allows existing, unaware programs to interact with patchgroups, in the same way that programs do not have to know about pipes for the shell to connect them in a pipeline. For example, a `depend` program could apply patchgroups to unmodified applications by setting up the patchgroups before calling `exec`. The following command line would ensure that `in` is not removed until all changes in the preceding `sort` have committed to disk:

```
depend "sort < in > out" "rm in"
```

Patchgroup support is implemented by an L2FS module that lives above a file system module like ext2. Each patchgroup corresponds to a pair of containing empty patches, and inter-patchgroup dependencies to dependencies between these empty patches. The L2FS module ensures that all file system changes are inserted between engaged patchgroups' empty patches. Figure 6 shows an example patch arrangement for two patchgroups.

While these simple dependencies are all that is necessary when using soft updates as the underlying consistency model, some extra work is required in the journal module to support patchgroups. When using a journal, writing the commit record atomically commits a transaction; additional patchgroup-specified dependencies for the data in each transaction must be shifted to the commit record itself. These dependencies then collapse into harmless dependencies from the commit record to itself or to previous commit records. Finally, for a metadata journal, blocks which are modified as part of

[2]These rules are really just a strictly enforced version of the requirement from §4.1 that all dependencies must be specified up-front.

a patchgroup must also be included in the transaction, making the metadata journal act like a full journal for those data blocks.

5.2 Case Studies

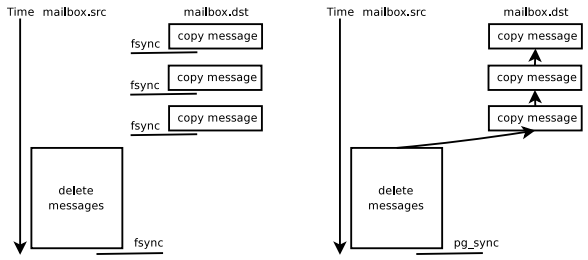
We studied the patchgroup interface by adding patchgroup support to three applications: the gzip compression utility, the Subversion version control client, and the UW IMAP mail server daemon. These applications were chosen for their relatively simple and explicit consistency requirements; we intended to test how well patchgroups implement existing application requirements, not to create new mechanisms. One effect of this choice is that, after some modification, versions of these applications could attain similar consistency guarantees if run on a fully-journaled file system with a conventional API. Patchgroups, however, make the required guarantees explicit, can be implemented on other types of file system, and introduce no additional cost on fully-journaled systems.

Gzip Our modified gzip uses patchgroups to make the input file's removal depend on the output file's data being written; thus, a crash cannot lose both files. The update adds 10 lines of code to gzip v1.3.9: simple consistency requirements are simple to add.

Subversion The Subversion version control system's client [2] manipulates a local working copy of a repository. The working copy library is designed to avoid data corruption or loss should the process exit prematurely from a working copy operation. This safety is achieved using application-level write ahead journaling, where each entry in Subversion's journal is either idempotent or atomic. Depending on the file system, however, even with this precaution a working copy operation may not be protected against a crash. For example, the journal file is marked as complete by moving it from its temporary to live location. Should the file system completely commit the file rename before the file data, and crash before completing the file data commit, then a subsequent journal replay could corrupt the working copy.

The working copy library could ensure a safe commit ordering by syncing files as necessary. The Subversion server (repository) library takes this approach, but the Subversion developers deemed this approach too slow to be worthwhile at the client [24]. Instead, the working copy library assumes that first, all preceding writes to a file's data are committed before the file is renamed; and second, metadata updates are effectively committed in their system call order. In addition to being obtuse, file renaming does not behave as required on many systems; for example, neither NTFS with journaling nor BSD UFS with soft updates provide the required properties. The Subversion developers essentially specialized their consistency mechanism for one file system, ext3 in either "ordered" or full journaling mode.

We updated the Subversion working copy library to use patchgroups to express commit ordering requirements without relying on properties of the underlying file system implementation. The use of the file rename property is replaced in two ways. First, files created in a temporary location and then moved into their live location (e.g. directory status and journal files) now make the rename depend on the file data writes. Second, files only referenced by live files (e.g. updated file copies used by journal file entries) can live with a weaker ordering; the data for these files need only precede the installation of the referencing file. The use of linearly ordered metadata updates is also replaced by patchgroup dependencies. Although the original operations assumed a linear ordering of metadata updates, the actual order requirements are considerably less strict. For example, the updated file copies used by the journal may be committed independently of each other. Furthermore, most journal playback operations may commit independently of each other; only interacting operations, such as a file read and subsequent rename, need ordering.



a) Unmodified, `fsync` **b)** Patchgroups

Figure 7: UW IMAP server, without and with patchgroups, moving three messages from mailbox.src to mailbox.dst.

Once we understood Subversion v1.4.3's requirements, it took a day to add the 220 lines of code that enforce safety for conflicted updates (out of 25,000 in the working copy library).

UW IMAP We updated the University of Washington's IMAP mail server (v2004g) [3] to ensure mail updates are committed to disk in safe and more efficient orderings. The Internet Message Access Protocol (IMAP) [6] provides remote access to a mail server's email message store. The most relevant IMAP commands synchronize changes to the server's disk (CHECK), copy a message from the selected mailbox to another mailbox (COPY), and delete messages marked for deletion (EXPUNGE).

We updated the imapd and mbox mail storage drivers to use patchgroups, ensuring that all disk writes occur in a safe ordering without enforcing a specific block write order. The original server conservatively preserved command ordering by syncing the mailbox file after each CHECK on it or COPY into it. For example, Figure 7a illustrates moving messages from one mailbox to another. With patchgroups, each command's file system updates are executed under a distinct patchgroup and, through the patchgroup, made to depend on the previous command's updates. This is necessary, for example, so that moving a message to another folder (accomplished by copying to the destination file and then removing from the source file) cannot lose the copied message should the server crash part way through the disk updates. The updated CHECK command uses `pg_sync` to sync all preceding disk updates. This enhancement to CHECK removes the requirement that COPY sync its destination mailbox: the client's CHECK request will ensure changes are committed to disk, and the patchgroup dependencies ensure changes are committed in a safe ordering. Figure 7b illustrates using patches to move messages.

These changes improve UW IMAP by ensuring ensure disk write ordering correctness and by performing disk writes more efficiently. As each command's changes now depend on the preceding command's changes, it is no longer required that all code specifically ensure its changes are committed before any later, dependent command's changes. Without patchgroups, modules like the mbox driver forced a conservative disk sync protocol because ensuring safety more efficiently required additional state information, adding further complexity. The Dovecot IMAP server's source code notes this exact difficulty [1, maildir-save.c]:

```
/* FIXME: when saving multiple messages, we could get
   better performance if we left the fd open and
   fsync()ed it later */
```

The performance of the patchgroup-enabled UW IMAP mail server is evaluated in Section 8.5.

6 MODULES

A Featherstitch configuration is composed of many modules that cooperate to implement file system functionality. There are three major types of modules. Closest to the disk are block device (BD) modules, which have a fairly conventional block device interface with interfaces such as "read block" and "flush". Closest to the system call interface are *common file system* (CFS) modules, which have an interface similar to VFS [14]. In between these interfaces are modules implementing a *low-level file system* (L2FS) interface, which helps divide file system implementations into code common across block-structured file systems and code specific to a given file system layout. The L2FS interface has functions to allocate blocks, add blocks to files, allocate file names, and other file system micro-operations. A generic CFS-to-L2FS module called UHFS ("universal high-level file system") decomposes familiar VFS operations like write, read, and append into L2FS micro-operations. File system implementations, such as our ext2 and UFS implementations, generally use the L2FS interface.

Module interfaces include patches explicitly, allowing modules to examine and modify dependencies. For instance, every L2FS function that might modify the file system takes a `patch_t **p` argument. Before the function is called, `*p` should be set to the patch, if any, on which the modification should depend; when the function returns, `*p` is set to some patch corresponding to the modification itself. For example, this function is called to append a block to an L2FS inode `f`:

```
int (*append_file_block)(LFS_t *module,
    fdesc_t *f, uint32_t block, patch_t **p);
```

6.1 ext2 and UFS

Featherstitch currently has modules that implement two file system types, Linux ext2 and 4.2 BSD UFS (Unix File System, the modern incarnation of the Fast File System [18]). Both of these modules initially generate dependencies arranged according to the soft updates rules; other dependency arrangements are achieved by transforming these. To the best of our knowledge, our implementation of ext2 is the first to provide soft updates consistency guarantees.

Both modules are implemented at the L2FS interface. Unlike FreeBSD's soft updates implementation, once these modules set up the desired dependencies for the patches they create, they no longer need to concern themselves with those patches—the block device subsystem will track and enforce the dependencies.

6.2 Journal

The journal module sits below a regular file system, such as ext2, and transforms incoming patches into patches implementing journal transactions. File system blocks are copied into the journal; a commit record depends on the journal patches; and the original file system patches depend in turn on the commit record. Any soft updates-like dependencies among the original patches are removed, since they are not needed when the journal handles consistency; however, the journal does care to ensure that user-specified dependencies, such as patchgroups, are not violated. The journal format is similar to ext3's [32]: a transaction contains a list of block numbers, the data to be written to those blocks, and finally a single commit record. Although the journal modifies existing patches' direct dependencies, it ensures that the new dependencies will never introduce a block-level cycle.

Like ext3, transactions are required to commit in sequence. Therefore the journal module sets each commit record to depend on the previous commit record, and each completion record to depend on the previous completion record. This allows multiple outstanding transactions in the journal, which benefits performance,

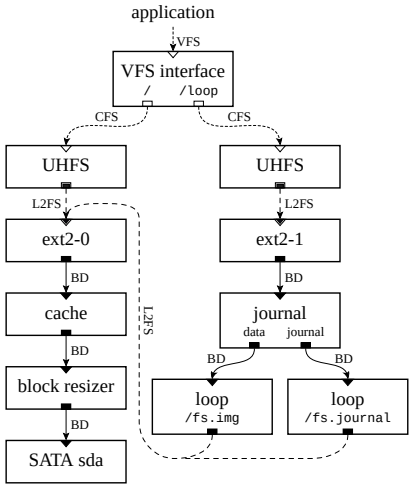


Figure 8: A running Featherstitch configuration. */* is a soft updated file system on an IDE drive; */loop* is an externally journaled file system on loop devices.

but ensures that in the event of a crash, the journal will contain only contiguous sequential transactions.

Since the commit record is created at the end of the transaction, the journal module uses a special empty patch explicitly held in memory to prevent file system changes from being written to the disk until the transaction is complete. This empty patch is set to depend on the previous transaction's completion record, which prevents merging between transactions while allowing merging within a transaction. This temporary dependency is removed when the real commit record is created and its dependencies are set up as described above.

Our journal module prototype can run in full data journal mode, where every updated block is written to the journal, or in metadata-only mode, where only blocks containing file system metadata are written to the journal. It can tell which blocks are which by looking for a special flag on each patch set by the UHFS module.

We provide several other modules that modify dependencies, including an "asynchronous mode" module that removes all dependencies, allowing the buffer cache to write blocks in any order. This implements similar semantics as existing file systems like ext2 in asynchronous write mode.

6.3 Buffer Cache

The Featherstitch buffer cache both caches blocks in memory and ensures that modifications are written to stable storage in a safe order. Modules "below" the buffer cache—that is, between its output interface and the disk—can reorder block writes at will without violating dependencies, since block writes below the buffer cache contain only in-flight patches. The buffer cache sees the complex consistency protocols that other modules want to enforce as nothing more than sets of dependencies among patches; it has no idea what consistency protocols it is implementing, if any.

Our prototype buffer cache module uses a modified FIFO policy to write dirty blocks and an LRU policy to evict clean blocks. (Upon being written, a dirty block becomes clean and may then be evicted.) The FIFO policy used to write blocks is modified only to preserve the in-flight safety property: a block will not be written if none of its patches are ready to write. Once the cache finds a block with ready patches, it extracts all ready patches P from the block, reverts any remaining patches on that block, and sends the resulting data to the disk driver. The ready patches are marked in-flight and

will be committed when the disk driver acknowledges the write. The block itself is also marked in flight until the current version commits, ensuring that the cache will wait until then to write the block again.

As a performance heuristic, when the cache finds a writable block n, it then checks to see if block $n+1$ can be written as well. It continues writing increasing block numbers until some block is either unwritable or not in the cache. This simple optimization greatly improves I/O wait time, since the I/O requests are merged and reordered in Linux's elevator scheduler. Nevertheless, there may still be important opportunities for further optimization: for example, since the cache will write a block even if only one of its patches is ready, it can choose to revert patches unnecessarily when a different order would have required fewer writes.

6.4 Loopback

The Featherstitch loopback module demonstrates how pervasive support for patches can implement previously unfamiliar dependency semantics. Like Linux's loopback device, it provides a block device interface that uses a file in some other file system as its data store; unlike Linux's block device, consistency requirements on this block device are obeyed by the underlying file system. The loopback module preserves incoming dependencies and forwards them to the underlying data store. As a result, the data store will honor those dependencies and preserve the loopback file system's consistency, even if the data store would normally provide no guarantees for consistency of file system data (e.g., it used metadata-only journaling).

Figure 8 shows an albeit contrived example configuration using the loopback module. A file system image is mounted with an external journal, both of which are loopback block devices stored on the root file system (which uses soft updates). The journaled file system's ordering requirements are sent through the loopback module as patches, allowing dependency information to be maintained across boundaries that might otherwise lose that information. In contrast, without patches and the ability to forward patches through loopback devices, BSD cannot express soft updates' consistency requirements through loopback devices. The modules in Figure 8 are a complete Featherstitch configuration.

7 Implementation

The Featherstitch prototype implementation runs as a Linux 2.6 kernel module. It interfaces with the Linux kernel at the VFS layer and the generic block device layer. In between is a Featherstitch module graph that replaces Linux's conventional file system layers and buffer cache. A small kernel patch informs Featherstitch of process fork and exit events as required to update per-process patchgroup state.

During initialization, the Featherstitch kernel module registers a VFS file system type with Linux. Each file system Featherstitch detects on a specified disk device can then be mounted from Linux using a command like `mount -t kfs kfs:`*`name`* `/mnt/point`. Since Featherstitch provides its own patch-aware buffer cache, it sets `O_SYNC` on all opened files as the simplest way to bypass the normal Linux cache and ensure that the Featherstitch buffer cache obeys all necessary dependency orderings.

Featherstitch modules interact with Linux's generic block device layer mainly via `generic_make_request`. This function sends read or write requests to a Linux disk scheduler, which may reorder and/or merge the requests before eventually releasing them to the device. Writes are considered in flight as soon as they are enqueued on the disk scheduler. A callback notifies Featherstitch when the disk controller reports request completion; for writes, this commits the corresponding patches. The disk safety property requires that the disk controller wait to report completion until a write has reached stable storage. Most drives instead report completion when a write has reached the drive's volatile cache. Ensuring the stronger property could be quite expensive, requiring frequent barriers or setting the drive cache to write-through mode; either choice seems to prevent older drives from reordering requests. The solution is a combination of SCSI tagged command queuing (TCQ) or SATA native command queuing (NCQ) with either a write-through cache or "forced unit access" (FUA). TCQ and NCQ allow a drive to independently report completion for multiple outstanding requests, and FUA is a per-request flag that says the disk should report completion only after the request reaches stable storage. Recent SATA drives handle NCQ plus write-through caching or FUA exactly as we would want: the drive appears to reorder write requests, improving performance dramatically relative to older drives, but reports completion only when data reaches the disk. We use a patched version of the Linux 2.6.20.1 kernel with good support for NCQ and FUA, and a recent SATA2 drive.

Our prototype's Linux integration has several performance problems. For example, data blocks are often stored in Linux's buffer cache as well as our own, increasing memory pressure. Writing a block requires copying that block's data whether or not any patches were undone. Our buffer cache's size is more limited than Linux's, since it currently requires that all blocks are stored in permanently-mapped kernel memory.

8 Evaluation

We evaluate the effectiveness of patch optimizations, the performance of the Featherstitch implementation relative to Linux ext2 and ext3, the correctness of the Featherstitch implementation, and the performance of patchgroups. This evaluation shows that patch optimizations significantly reduce patch memory and CPU requirements; that a Featherstitch patch-based storage system has overall performance that is within reason, even though it is slower than Linux in some benchmarks; that Featherstitch file systems are consistent after system crashes; and that a patchgroup-enabled IMAP server performs as well as or better than the unmodified server.

8.1 Methodology

All tests were run on a Dell Precision 380 with a 3.2 GHz Pentium 4 CPU, 2 GB of RAM, and a Seagate ST3320620AS 320 GB 7200 RPM SATA2 disk. All tests use a 10 GB file system and the Linux 2.6.20.1 kernel with the Ubuntu v6.06.1 distribution. Because the Featherstitch-kernel integration does not use high memory and because Featherstitch uses its own block cache in addition to the kernel's, we limit Featherstitch and Linux configurations to an effective 312 MB of RAM available for the block cache. (Featherstitch is given more RAM total, roughly compensating for extra memory pressure due to Linux integration issues; see §7.) Only the PostMark benchmark is sensitive to cache size.

We examine Linux ext2 (in asynchronous mode) and ext3 (in writeback, ordered, and full journal modes), and Featherstitch ext2 (in asynchronous, soft updates, metadata journal, and full journal modes), all created with default configurations.

The three journal modes provide different consistency guarantees. The strength of these guarantees is strictly ordered:

$$\text{asynchronous} < \text{writeback} < \text{ordered} < \text{full}$$

Writeback journaling commits metadata atomically and commits data only after the corresponding metadata. Featherstitch metadata journaling is equivalent to ext3 writeback journaling. Ordered journaling commits data associated with a given transaction prior to

Optimization	# Patches	Undo data	System time
Untar test			
None	619,740	459.41 MB	3.33 sec
Hard patches	446,002	205.94 MB	2.73 sec
Overlap merging	111,486	254.02 MB	1.87 sec
Both	68,887	0.39 MB	1.83 sec
Delete test			
None	299,089	1.43 MB	0.81 sec
Hard patches	41,113	0.91 MB	0.24 sec
Overlap merging	54,665	0.93 MB	0.31 sec
Both	1,800	0.00 MB	0.15 sec
PostMark test			
None	4,591,628	3,175.28 MB	24.46 sec
Hard patches	2,544,300	1,583.75 MB	20.00 sec
Overlap merging	543,738	1,587.81 MB	12.57 sec
Both	624,776	0.10 MB	13.78 sec
Andrew test			
None	70,925	64.09 MB	4.33 sec
Hard patches	50,803	36.18 MB	4.31 sec
Overlap merging	12,466	27.91 MB	4.23 sec
Both	10,398	0.04 MB	4.30 sec

Figure 9: Effectiveness of Featherstitch optimizations.

the proceeding transaction's metadata, and is the most commonly used ext3 journaling mode. In all tests ext3 writeback and ordered journaling modes performed similarly, and Featherstitch does not implement ordered mode, so we report only writeback results. Full journaling commits data atomically.

There is a notable write durability difference between the default Featherstitch and Linux ext2/ext3 configurations: Featherstitch safely presumes a write is durable after it is on the disk platter, whereas Linux ext2 and ext3 by default presume a write is durable once it reaches the disk cache. However, Linux can write safely (by restricting the disk to providing only a write through cache) and Featherstitch can write unsafely (by disabling FUA). We distinguish safe (e.g., FUA/write-through cache) from unsafe results when comparing the systems. All timing results are the mean over five runs.

To evaluate patch optimizations and Featherstitch as a whole we ran four benchmarks. The *untar benchmark* untars and syncs the Linux 2.6.15 source code from the cached file `linux-2.6.15.tar` (218 MB). The *delete benchmark*, after unmounting and remounting the file system following the untar benchmark, deletes the result of the untar benchmark and syncs. The *PostMark benchmark* emulates the small file workloads seen on email and netnews servers [13]. We use PostMark v1.5, configured to create 500 files ranging in size from 500 B to 4 MB; perform 500 transactions consisting of file reads, writes, creates, and deletes; delete its files; and finally sync. The modified *Andrew benchmark* emulates a software development workload. The benchmark creates a directory hierarchy, copies a source tree, reads the extracted files, compiles the extracted files, and syncs. The source code we use for the modified Andrew benchmark is the Ion window manager, version 2-20040729.

8.2 Optimization Benefits

We evaluate the effectiveness of the patch optimizations discussed in Section 4, in terms of the total number of patches created, amount of undo data allocated, total amount of memory allocated for Featherstitch, and system CPU time used. Figure 9 shows these results for the untar, delete, PostMark, and Andrew benchmarks for Featherstitch ext2 in soft updates mode, with all combinations of using hard patches and overlap merging. The results for metadata and full data journaling are similar. Both optimizations work well alone, but their combination is particularly effective at reducing the amount of undo data—which, again, is pure overhead relative to conventional file systems. (However, the combined optimizations slightly increase CPU time relative to overlap merging alone in some benchmarks.) Undo data memory usage is reduced by at least 99.9%, the number of patches created is reduced by 85–99%, and system CPU time is reduced by up to 81%.

System	Untar	Delete	PostMark	Andrew
Featherstitch ext2				
soft updates	**6.4 [1.9]**	**0.8 [0.2]**	**47.0 [13.5]**	**36.9 [4.1]**
meta journal	**8.5 [2.6]**	**1.4 [0.5]**	**70.8 [15.6]**	**36.7 [4.2]**
full journal	**12.8 [3.2]**	**1.3 [0.5]**	**93.8 [19.7]**	**36.9 [4.4]**
async	4.3 [1.4]	0.7 [0.2]	45.4 [7.3]	36.4 [4.1]
full journal	11.1 [3.8]	1.1 [0.5]	83.9 [23.7]	36.7 [4.3]
Linux				
ext3 writeback	**19.6 [1.0]**	**4.5 [0.2]**	**55.5 [3.6]**	**37.6 [4.0]**
ext3 journal	**17.1 [1.2]**	**4.6 [0.3]**	**80.5 [4.7]**	**38.1 [4.0]**
ext2	9.2 [0.7]	4.5 [0.1]	45.4 [2.0]	36.8 [4.0]
ext3 journal	13.3 [1.2]	4.3 [0.2]	71.8 [4.7]	37.2 [4.0]

Figure 10: Benchmark times (seconds). System CPU times are in square brackets. Safe configurations are **bold**, unsafe configurations are normal text.

8.3 Benchmarks

We benchmark Featherstitch and Linux ext2/ext3 for the untar, delete, PostMark, and Andrew benchmarks. Results are listed in Figure 10. The general result is that Featherstitch performs comparably with Linux ext2/ext3 when providing similar durability guarantees. Linux ext2/ext3 sometimes outperforms Featherstitch (particularly for journaling modes and the PostMark test), and sometimes the reverse.

While Featherstitch performs similarly for total time for the untar, delete, and Andrew benchmarks, during each test Featherstitch uses significantly more CPU time than Linux ext2 or ext3 do—up to three times the CPU time. Higher CPU requirements are an important concern and, despite much progress in our optimization efforts, remain a weakness. Some of the contributors to Featherstitch CPU usage are inherent, such as patch creation, while others are artifacts of the current implementation, such as creating a second copy of a block to write it to disk; we have not separated these two categories.

8.4 Correctness

In order to check that we had implemented the journaling and soft updates rules correctly, we implemented a Featherstitch module which crashes the operating system, without giving it a chance to synchronize its buffers, at a random time during each run. In Featherstitch asynchronous mode, after crashing, fsck nearly always reported that the file system contained many references to inodes that had been deleted, among other errors: the file system was corrupt. With our soft updates dependencies, the file system was always soft update consistent: fsck reported that inode reference counts were higher than the correct values (an expected discrepancy after a soft updates crash). With journaling, fsck always reported that the file system was consistent after the journal replay.

8.5 Patchgroups

We evaluate the performance of the patchgroup-enabled UW IMAP mail server (§5.2) by benchmarking moving 1,000 messages from one folder to another. To move the messages, the client selects the source mailbox (containing 1,000 2 kB messages), creates a new mailbox, copies each message to the new mailbox and marks each

File System	Consistency method	Time	Writes
Featherstitch ext2			
soft updates	fsync	65.2 [0.3]	8,083
full journal	fsync	50.9 [0.4]	7,114
soft updates	patchgroups	27.4 [1.1]	3,015
full journal	patchgroups	1.4 [0.4]	32
Linux ext3			
journal	fsync	19.9 [0.3]	2,531
journal	assume linear	1.3 [0.3]	26

Figure 11: IMAP benchmark: move 1,000 messages. Times are in seconds (system CPU times in square brackets) and writes in number of requests.

source message for deletion, expunges the marked messages, commits the mailboxes, and logs out.

Figure 11 shows the results for safe file system configurations, reporting total time, system CPU time, and the number of disk write requests (an indicator of the number of required seeks in safe configurations). We benchmark Featherstitch and Linux with the unmodified server (sync after each message copy), Featherstitch with the patchgroup-enabled server (sync only at the end), and Linux and Featherstitch with the server modified to assume and take advantage of fully journaled file systems (changes are effectively committed linearly, so sync only at the end).

Figure 11 lists only safe configurations; unsafe Featherstitch and Linux configurations complete in about 1.5 seconds. Featherstitch meta and full journal modes perform similarly so we report only for the full journal mode. Linux ext3 writeback, ordered, and full journal modes also perform similarly, so we report only for the full journal mode. Using the linear consistency method with Featherstitch full journaling performs similarly to the corresponding case with Linux fill journaling.

Fully journaled Featherstitch with patchgroups performs at least as well as all other (safe and unsafe) Featherstitch and all Linux configurations. It is 11–13 times faster than safe Linux ext3 with the unmodified server. Using the linear consistency method with Featherstitch and Linux, the IMAP server performs similarly to the patchgroup consistency method with Featherstitch. However, the linear consistency method requires the underlying file system use full data journaling.

Featherstitch in soft updates mode with patchgroups is significantly slower than Featherstitch in journal modes. As the number of write requests indicate, each of soft updates mode's patchgroups require multiple write requests (e.g. to update the destination mailbox and the destination mailbox's modification time). In contrast, journaling is able to commit a large number of copies atomically, using only a small constant number of requests. Featherstitch with the unmodified server makes significantly many more requests than Linux ext3; we believe this is due to Featherstitch also committing file system summary information that Linux does not commit for the `fsync` system call. Featherstitch with patchgroups also currently has this limitation.

8.6 Summary

We find that our optimizations greatly reduce system overheads, including undo data and system CPU time; that Featherstitch obtains reasonable performance on several benchmarks, comparable with Linux in most cases despite the additional effort required to maintain patches; that CPU time remains an optimization opportunity; that applications can effectively define consistency requirements with patchgroups that apply to many file systems; and that the Featherstitch implementation correctly implements soft updates and journaling consistency. Our results are encouraging, indicating that even a patch-based prototype can implement different consistency models with reasonable cost.

9 Conclusion

Featherstitch patches provide a new way for file system implementations to formalize the "write-before" relationship among buffered changes to stable storage. Thanks to several optimizations, the performance of our prototype is usually at least as fast as Linux when configured to provide similar consistency guarantees, although in some cases it still requires improvement. Patches simplify the implementation of consistency mechanisms like journaling and soft updates by separating the specification of write-before relationships from their enforcement. Using patches also allows our prototype to be divided into modules that cooperate loosely to implement strong consistency guarantees. Additionally, patches can be extended into userspace, allowing applications to specify more precisely what their specific consistency requirements are. This provides the buffer cache more freedom to reorder writes without violating the application's needs, while simultaneously freeing the application from having to micromanage writes to disk. We present results for an IMAP server modified to take advantage of this feature, and show that it can significantly reduce both the total time and the number of writes required for our benchmark.

There are several areas in which we would like to improve our work. The obvious first area we would like to work on is the performance of our Featherstitch prototype. We have already improved the performance by at least five orders of magnitude over the original implementation, but it is still not as fast as it could be. We would also like to explore a variety of extensions to Featherstitch, such as sending patches over network file systems and implementing other consistency mechanisms like shadow paging. Finally, we would like to try using patchgroups for several more complicated applications, like databases, to see how well they fit the needed semantics and how well they perform.

Acknowledgments

We would like to thank the members of our lab at UCLA, "TERTL", for many useful discussions about this work, and for reviewing drafts of this paper. In particular, Steve VanDeBogart provided extensive help with both Linux kernel details and drafts. Further thanks go to Liuba Shrira, who provided sustained encouraging interest in the project, and Stefan Savage for early inspiration. Our shepherd, Andrea Arpaci-Dusseau, and the anonymous reviewers provided very useful feedback and guidance. Our work on Featherstitch was supported by the National Science Foundation under Grant Nos. 0546892 and 0427202. Additionally, Christopher Frost and Mike Mammarella were awarded SOSP student travel scholarships, supported by the National Science Foundation, to present this paper at the conference.

References

[1] *Dovecot*. Version 1.0 beta7, `http://www.dovecot.org/`.

[2] *Subversion*. `http://subversion.tigris.org/`.

[3] *UW IMAP toolkit*. `http://www.washington.edu/imap/`.

[4] N. C. Burnett. *Information and Control in File System Buffer Management*. PhD thesis, University of Wisconsin-Madison, July 2006.

[5] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 19–28, June 2004.

[6] M. Crispin. Internet Message Access Protocol - version 4rev1. RFC 3501, IETF, March 2003.

[7] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided Resynchronization for Software RAID. In *Proceedings of the Fourth USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[8] E. Gal and S. Toledo. A transactional Flash file system for microcontrollers. In *Proceedings of the 2005 USENIX Annual Technical Conference*, pages 89–104, Anaheim, California, Apr. 2005.

[9] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems*, 18(2):127–153, May 2000.

[10] J. S. Heidemann and G. J. Popek. A Layered Approach to File System Development. Technical report, UCLA, Los Angeles, CA, March 1991.

[11] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Transactions on Computer Systems*, 12(1): 58–89, Feb. 1994.

[12] H. Huang, W. Hung, and K. G. Shin. FS2: Dynamic Data Replication in Free Disk Space for Improving Disk Performance and Energy Consumption. In *Proceedings of 20th ACM Symposium on Operating Systems Principles*, pages 263–276, October 2005.

[13] J. Katcher. PostMark: A new file system benchmark. TR0322, Network Appliance, 1997. `http://tinyurl.com/27ommd`.

[14] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the 1986 USENIX Summer Technical Conference*, pages 238–47, Atlanta, Georgia, 1986.

[15] B. Liskov and R. Rodrigues. Transactional file systems can be fast. In *Proceedings of the 11th ACM SIGOPS European Workshop*, Leuven, Belgium, Sept. 2004.

[16] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems (TOCS)*, 12(2):123–164, 1994. ISSN 0734-2071.

[17] M. K. McKusick and G. R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the Fast Filesystem. In *Proceedings of the 1999 USENIX Technical Conference—FREENIX Track*, pages 1–17, Monterey, California, June 1999.

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

[19] K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A Versatile and User-Oriented Versioning File System. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 115–128, San Francisco, CA, March/April 2004. USENIX Association.

[20] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, Brighton, England, Oct. 2005.

[21] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the Sync. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–14, Seattle, WA, November 2006.

[22] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies*, Monterey,CA, 2002.

[23] D. S. H. Rosenthal. Evolving the vnode interface. In *Proceedings of the 1990 USENIX Summer Technical Conference*, Anaheim, California, Jan. 1990.

[24] M. Rowe. wc atomic rename safety on non-ext3 file systems. Subversion developer mailing list, March 2007. `http://svn.haxx.se/dev/archive-2007-03/0064.shtml`.

[25] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[26] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-Safe Disks. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, November 2006.

[27] M. Sivathanu, V. Prabhakaran, F. Popovici, T. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, California, Mar. 2003.

[28] M. Sivathanu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A Logic of File Systems. In *Proceedings of the Fourth USENIX Symposium on File and Storage Technologies (FAST '05)*, San Francisco, CA, December 2005.

[29] M. Sivathanu, L. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, California, Dec. 2005.

[30] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. In *Proceedings of the 1993 USENIX Summer Technical Conference*, pages 161–174, Cincinnati, OH, 1993.

[31] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, San Francisco, California, Mar. 2003.

[32] S. Tweedie. Journaling the Linux ext2fs Filesystem. In *LinuxExpo '98*, 1998.

[33] M. Vilayannur, P. Nath, and A. Sivasubramaniam. Providing tunable consistency for a parallel file store. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, California, Dec. 2005.

[34] M. Waychison. fallocate support for bitmap-based files. linux-ext4 mailing list, June 2007. `http://www.mail-archive.com/linux-ext4@vger.kernel.org/msg02382.html`.

[35] C. P. Wright, M. C. Martino, and E. Zadok. NCryptfs: A Secure and Convenient Cryptographic File System. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 197–210, San Antonio, Texas, June 2003.

[36] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, D. P. Quigley, E. Zadok, and M. N. Zubair. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage*, Mar. 2006.

[37] J. Yang, P. Twohey, D. Engler, and M. Musuvathni. Using model checking to find serious file system errors. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, San Francisco, California, Dec. 2004.

[38] J. Yang, C. Sar, and D. Engler. EXPLODE: a lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Seattle, Washington, Nov. 2006.

[39] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 55–70, June 2000.

[40] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. In *Proceedings of the 1999 USENIX Annual Technical Conference*, pages 57–70, Monterey, California, June 1999.