CS240, Adv OS
Spring 2011
Engler

# 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 in part on "write-before" relationships where some changes to stable storage are delayed until other changes commit. A journaled file system, for example, must commit a 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 abstraction, the *patch*, that makes write-before relationships explicit and file system agnostic. A patch-based file system implementation expresses dependencies among writes, leaving lower system layers to determine write orders that satisfy those dependencies. Storage system modules can examine and modify the dependency structure, and generalized file system dependencies are naturally exportable to user level. Our patch-based storage system, *Featherstitch*, includes several important optimizations that reduce patch overheads by orders of magnitude. Our ext2 prototype runs in the Linux kernel and supports asynchronous writes, soft updates-like dependencies, and journaling. It outperforms similarly reliable ext2 and ext3 configurations on some, but not all, benchmarks. It also supports unusual configurations, such as correct dependency enforcement within a loopback file system, and lets applications define 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

Write-before relationships, which require that some changes be committed to stable storage before others, underlie every mechanism for ensuring file system consistency and reliability from journaling to synchronous writes. Featherstitch is a complete storage system built on a concrete form of these relationships, a simple, uniform, and file system agnostic data type called the *patch*. Featherstitch's API design and performance optimizations make patches a promising implementation strategy as well as a useful abstraction.

A patch represents both a change to disk data and any *dependencies* between that change and other changes. Patches were initially inspired by BSD's soft updates dependencies [9], but whereas soft updates implement a particular type of consistency and involve many structures specific to the UFS file system [18], 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 patch dependency structures, thus defining consistency policies for the underlying storage system to follow.

A uniform and pervasive patch abstraction may simplify implementation, extension, and experimentation for file system consistency and reliability mechanisms. File system implementers currently find it difficult to provide consistency guarantees [18, 34] and implementations are often buggy [40, 41], a situation further complicated by file system extensions and special disk interfaces [5, 20, 23, 28, 30, 32, 38]. File system extension techniques such as stackable file systems [10, 24, 42, 43] 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, 29], 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, 16, 37]. Some users, however, implement their own end-to-end reliability for some data and prefer to avoid *any* consistency slowdowns in the file system layer [36]. Patches can represent all these choices, and since they provide a common language for file systems and extensions to discuss consistency requirements, even combinations of consistency mechanisms can 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` or use tedious and fragile sequences of operations that assume particular file system consistency semantics. *Patchgroups*, our example user-level patch interface, export to applications some of patches' benefits for kernel file system implementations and extensions. Modifying an IMAP mail server to use patchgroups required only localized changes. The result both meets IMAP's consistency requirements on any reasonable patch-based file system and avoids the performance hit of full synchronization.

"?"

Production file systems use system-specific optimizations to achieve consistency without sacrificing performance; we had to improve performance in a general way. A naive patch-based storage system scaled terribly, spending far more space and time on dependency manipulation than conventional systems. However, optimizations reduced patch memory and CPU overheads significantly. A PostMark test that writes approximately 3.2 GB of data allocates

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*SOSP'07,* October 14–17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010 . . . $5.00.

FS work driven by two first order problems: (1) seek so slow that tiny reduction = huge effect, (2) handle crashes = hard. crash ~ context switch: after crash FS starts running w/ disk memory (shared mem) in some messed up, half-way state. has to push forward or backward to a clean legal state. harder than concurrency since this context switch (crash) can happen at any program point and you can't prevent it - with threads can always acquire a lock.

75 MB of memory throughout the test to store patches and soft updates-like dependencies, less than 3% of the memory used for file system data and about 1% of that required by unoptimized Featherstitch. Room for improvement remains, particularly in system time, but Featherstitch outperforms equivalent Linux configurations on many of our benchmarks; it is at most 30% slower on others.

cpu or I/O the bottleneck?

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.

In this paper, 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 into pluggable modules, hopefully making it configurable, extensible, and relatively easy to understand. Finally, our evaluation compares Featherstitch and Linux-native file system implementations, and 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 [26]. 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 [40], avoids time-consuming file system checks on post-crash reboot 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.

Patches naturally represent both journaling and soft updates, and we use them as running examples throughout the paper. In each case, our 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 [18]. 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.

Although patches were designed to represent any write-before relationship, implementations of other consistency mechanisms, such as shadow paging-style techniques for write anywhere file layouts [11] or ACID transactions [37], are left for future work.

CAPFS [35] and Echo [17] 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 bring 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 among 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 user space 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 [21] and xsyncfs [22] to reduce the aggregate performance impact of strong consistency guarantees. Xsyncfs's *external synchrony* provides users with the same consistency guarantees as synchronous writes. Application writes are not synchronous, however. They are committed in groups using a journaling design, but additional write-before relationships are enforced on *non-file system* communication: a journal transaction must commit before output from any process involved in that transaction becomes externally visible via, for example, the terminal or a network connection. Dependency relationships are tracked across IPC as well. Featherstitch patches could be used to link file system behavior and xsyncfs process dependencies, or to define cross-network dependencies as in BlueFS; this would remove, for instance, xsyncfs's reliance on ext3. Conversely, Featherstitch applications could benefit from the combination of strict ordering and nonblocking writes provided by xsyncfs. Like xsyncfs, stackable module software for file systems [10, 24, 31, 38, 39, 42, 43] and other extensions to file system and disk interfaces [12, 27] might benefit from a patch-like mechanism that represented write-before relationships and consistency requirements agnostically.

Some systems have generalized a *single* consistency mechanism. Linux's ext3 attempted to make its journaling layer a reusable component, although only ext3 itself uses it. XN enforces a variant of soft updates on any associated library file system, but still requires that those file systems implement soft updates again themselves [13].

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 that stable storage commits data in units called **blocks**. All writes affect one or more blocks, and it is impossible to selectively write part of a block. In disk terms, a block is a sector or, for file system convenience, a few contiguous sectors.

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

| | |
|---|---|
| $p$ | a patch |
| $blk[p]$ | patch $p$'s block |
| $\boldsymbol{C}, \boldsymbol{U}, \boldsymbol{F}$ | the sets of all committed, uncommitted, and in-flight patches, respectively |
| $\boldsymbol{C}_B, \boldsymbol{U}_B, \boldsymbol{F}_B$ | 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 $\exists x : q \rightsquigarrow x \rightarrow p$) |
| $ddep[p]$ | $p$'s direct dependencies: $\{x \mid p \rightarrow x\}$ |

**Figure 1**: Patch notation.

meaning in the process of being written to disk. The intermediate in-flight state models reordering and delay in lower storage layers; for example, modern disks often cache writes to add opportunities for disk scheduling. 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 operating system receives this acknowledgment, it commits the relevant patches. Committed patches stay committed permanently, although their effects can be undone by subsequent patches. The sets $\boldsymbol{C}$, $\boldsymbol{U}$, and $\boldsymbol{F}$ represent all committed, uncommitted, and in-flight patches, respectively.

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

Disk controllers in this model write in-flight patches one block at a time, choosing blocks in an arbitrary order. In notation:

1. Pick some block $B$ with $\boldsymbol{F}_B \neq \varnothing$.
2. Write block $B$ and acknowledge each patch in $\boldsymbol{F}_B$.
3. Repeat.

Disks perform better when allowed to reorder requests, so operating systems try to keep many blocks in flight. A block write will generally put all of that block's uncommitted patches in flight, but a storage system may, instead, write a *subset* of those patches, leaving some of them in the uncommitted state. 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, where if the disk fails while a block $B$ is in flight, $B$ contains either the old data or the new data on recovery. Many journal designs do not require this, and include recovery procedures that handle in-flight block corruption—for instance, if the memory holding the new value of the block loses coherence before the disk stops writing [33]. Since patches model the write-before relationships underlying these journal designs, patches do not provide block atomicity themselves, and a patch-based file system with soft updates-like dependencies should be used in conjunction with a storage layer that provides 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 $q$ either after $p$ or at the same time as $p$. (Patches can be committed simultaneously only if they are on the same block.) A file system should create dependencies that express its desired consistency semantics. For example, a file system with 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 first if $p \rightsquigarrow q \rightsquigarrow p$. (Although a circular dependency chain entirely within a single block would be acceptable, 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 have also been committed:

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

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 the independence of in-flight blocks. This is precisely stated by the **in-flight safety property:**

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

This implies that $dep[\boldsymbol{F}_B] \cap dep[\boldsymbol{F}_{B'}] \subseteq \boldsymbol{C}$ for any $B' \neq B$, so the disk controller can write in-flight blocks in any order and still preserve disk safety. To uphold the in-flight safety property, the buffer cache must write blocks as follows:

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

The requirement that $\boldsymbol{F}_B = \varnothing$ ensures that at most one version of a block is in flight at any time. Also, the buffer cache must eventually write *all* dirty blocks, a liveness property.

The main Featherstitch implementation challenge is to design data structures that make it easy to create patches and quick to manipulate patches, and that help the buffer cache write blocks and patches according to the above procedure.

## 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 a subset of the dependencies called the *direct dependencies*. Each patch $p$ has a corresponding set of direct dependencies $ddep[p]$; we say $q$ *directly depends on* $p$, and write $q \rightarrow p$, when $p \in ddep[q]$. The dependency relation $q \rightsquigarrow p$ means that either $q \rightarrow p$ or $q \rightsquigarrow x \rightarrow p$ for some patch $x$.

Featherstitch maintains each block in its dirty state, including the effects of all uncommitted patches. However, 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. Some undo mechanism is required to break potential block-level dependency cycles, as shown in the next section. We considered alternate designs, such as maintaining a single "old" version of the block, but per-patch undo data gives file systems the maximum flexibility to create patch structures. However, many of our optimizations avoid storing unnecessary undo data, greatly reducing memory usage and CPU utilization.

Figure 1 summarizes our patch notation.

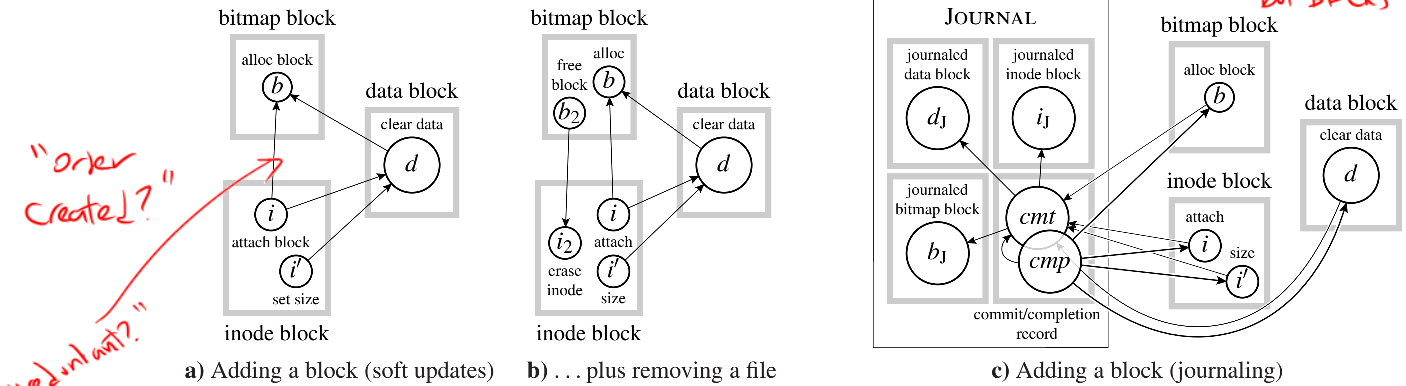



**Figure 2**: Example patch arrangements for an ext2-like file system. Circles represent patches, shaded boxes represent disk blocks, and arrows represent direct dependencies. **a)** A soft updates order for appending a zeroed-out block to a file. **b)** A different file on the same inode block is removed 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 patch implementations of 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 512 zero bytes to an empty file. The file system is based on Linux's ext2, an FFS-like file system with inodes and a free block bitmap. In such a file system, extending a file by one block 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 provides post-crash consistency without synchronous writes by tracking and obeying necessary dependencies among writes. 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 possible disk inconsistencies 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. Soft updates Rule 1 requires that $i \rightarrow b$. Rule 2 requires that $d$ depend on the nullification of previous pointers to the block; a simple, though more restrictive, way to accomplish this is to let $d \rightarrow b$, where $b$ depends on any such nullifications (there are none here). The dependencies $i \rightarrow d$ and $i' \rightarrow d$ provide an additional guarantee above and beyond metadata consistency, namely that no file ever contains accessible uninitialized data. Unlike Featherstitch, 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 [18].

Figure 2b shows how an additional file system operation can induce a circular dependency among blocks. Before the changes in Figure 2a commit, the user deletes a one-block file whose data block and inode happen to lie on the bitmap and inode blocks used by the previous operation. Rule 2 requires the dependency $b_2 \rightarrow i_2$; but given this dependency and the previous $i \rightarrow b$, neither the bitmap block nor the inode block can be written first! Breaking the cycle requires rolling back one or more patches, which in turn requires undo data. For example, the system might roll back $b_2$ and write the resulting bitmap block, which contains only $b$. Once this write commits, all of $i$, $i'$, and $i_2$ are safe to write; and once *they* 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 Patch Implementation

Our Featherstitch file system implementation creates patch structures corresponding directly to this abstraction. Functions like `patch_create_byte` create patches; 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 type of dependency. If two patches $q$ and $p$ affect the same block and have overlapping data ranges, and $q$ was created after $p$, then Featherstitch adds an *overlap dependency* $q \rightarrow p$ to ensure that $q$ is written after $p$. File systems need not detect such dependencies themselves.

For each block $B$, Featherstitch maintains a list of all patches with $blk[p] = B$. However, committed patches are not tracked; when patch $p$ commits, Featherstitch destroys $p$'s data structure and removes all dependencies $q \rightarrow p$. Thus, a patch whose dependencies have all committed appears like a patch with no dependencies at all. Each patch $p$ maintains doubly linked lists of its direct dependencies and "reverse dependencies" (that is, all $q$ where $q \rightarrow p$).

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 $q_i \rightarrow e$ and $e \rightarrow p_j$. This is useful for patchgroups; see Section 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 uses empty patches infrequently, and in the rest of this paper, patches are nonempty unless explicitly stated.

### 3.6 Discussion

The patch abstraction places only one substantive restriction on its users, namely, that circular dependency chains are errors. This restriction arises from the file system context: 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 dependency system that modeled abstract transactions. We chose write-before relationships as our foundation since they minimally constrain file system disk layout.

## 4 Patch Optimizations

Figure 3a shows the patches generated by a naive Featherstitch implementation when an application appends 16 kB of data to an existing empty file using four 4 kB writes. The file system is ext2 with soft updates-like dependencies and 4 kB blocks. Four blocks are allocated (patches $b_1$–$b_4$), written ($d_1$–$d_4$ and $d'_1$–$d'_4$), and attached to the file's inode ($i_1$–$i_4$); the inode's file size and modification time are updated ($i'_1$–$i'_4$ and $i''$); and changes to the "group descriptor" and superblock account for the allocated blocks ($g$ and $s$). Each application write updates 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, since they will *never* need to be reverted. And in larger examples, the effects are even worse. For example, when 256 MB of blocks are allocated in the untar benchmark described in Section 8, unoptimized Featherstitch allocates an additional 533 MB, mostly for patches and undo data.

This section presents optimizations based on generic dependency analysis that reduce these 23 patches and 33,000 bytes of undo data 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; for instance, they reduce memory overhead in the untar benchmark from 533 MB to just 40 MB.

### 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, and Featherstitch does not allocate any. This makes $p$ a **hard patch**: a patch without undo data. The system aims to reduce memory usage by making most patches hard. The challenge is to detect such patches without an oracle for future dependencies.

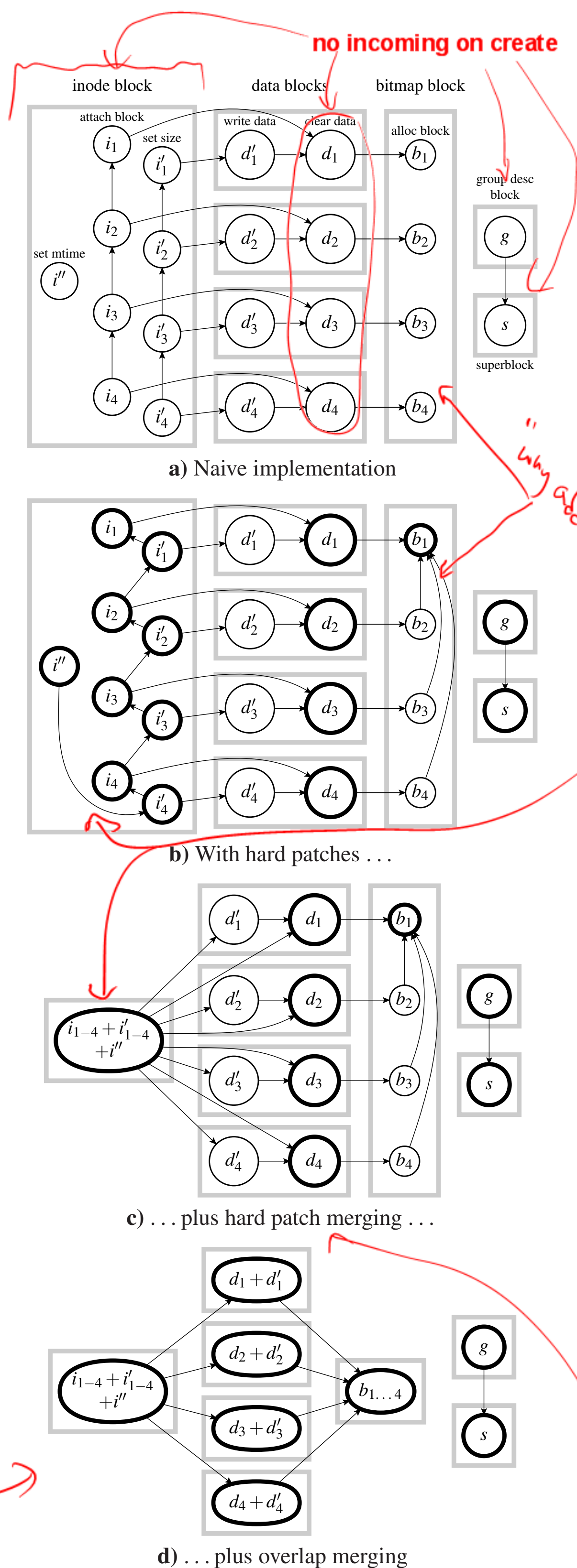
(Since a hard patch $h$ cannot be rolled back, any other patch on its block effectively depends on it. We represent this explicitly using, for example, overlap dependencies, and as a result, the buffer cache will write all of a block's hard patches whenever it writes the block.)

We now characterize one type of patch that 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 are on the same block $blk[p_n] = blk[p_1]$, and at least one patch in the middle is on a different block $blk[p_i] \neq blk[p_1]$. The patch $p_n$ is called the *head* of the block-level cycle. Now assume that a patch $p \in \boldsymbol{U}$ is not the head of any block-level cycle. One can then show that the buffer cache can write at least one patch without rolling back $p$. This is trivially possible if $p$ itself is ready to write. If it is not, then $p$ must depend on some uncommitted patch $x$ on a different block. However, we know that $x$'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 $x$ has finite length, and therefore contains an uncommitted patch $y$ whose dependencies have all committed. (If $y$ has in-flight dependencies, simply wait for the disk controller to commit them.) Since $y$ is not on $p$'s block, the buffer cache can write $y$ 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. Its proof strategy has two parts. First, the Featherstitch API restricts the creation of block-level cycles by restricting the creation of dependencies: *a patch's direct dependencies are all supplied at creation time*. Once $p$ 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

[1]The actual rule is somewhat more flexible: modules may add new direct dependencies if they guarantee that those dependencies don't pro-

key: (1) the *only* way to make a patch hard is if no incoming edge to the block when patch created, (2) if p->q, then q *must* exist before p created. thus, can always work out possible write orders from the graph. one possible: s, g, b1, d1, i1 [note: these all hard b/c no incoming edge to block when created], d'1 [incoming edge=not hard], i'1, b2 [incoming], d2, d'2, ...], etc




**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.

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 $blk[p]$—that is, if for all $y \rightsquigarrow x$ with $x$ an uncommitted patch on $p$'s block, $y$ is also on $p$'s block. If no other block depends on $p$'s, then clearly $p$ can't head up a current 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 16 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 data and dependencies 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* be merged. Featherstitch ensures that each block contains at most one hard patch. If 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 $ddep[h] \leftarrow ddep[h] \cup ddep[p]$. This changes $h$'s direct dependency set after $h$ was created, 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 any cyclic dependencies during the merge. 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 using soft updates-like dependencies. 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 cycles with head $p$, and $p$ can be transformed into a hard patch. Featherstitch will make $p$ hard by dropping its undo data, then merge $h$ into $p$. Although this type of merging is not very common in practice, it is necessary to preserve useful invariants, such as that no hard patch has a dependency on the same block.

## 4.3 Overlap Merging

The final type of merging combines soft patches with other patches, hard or soft, when they overlap. Metadata blocks, such as bitmap blocks, inodes, and directory data, tend to accumulate many nearby

duce 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.

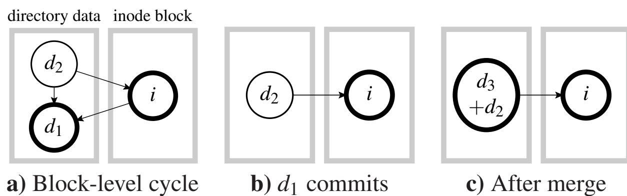



**Figure 4**: Soft-to-hard patch merging. **a)** Soft updates-like dependencies among directory data and an inode block. $d_1$ deletes a file whose inode is on $i$, so Rule 2 requires $i \rightarrow d_1$; $d_2$ allocates a file whose inode is on $i$, so Rule 1 requires $d_2 \rightarrow i$. **b)** Writing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

and overlapping patches as the file system gradually changes; for instance, Figure 3's $i_1$–$i_4$ all affect the same inode field. Even data blocks can collect overlapping dependencies. Figure 3's data writes $d'_n$ overlap, and therefore depend on, the initialization writes $d_n$—but our heuristic cannot make $d'_n$ hard since when they are created, dependencies exist from the inode block onto $d_n$. Overlap merging can combine these, and many other, mergeable patches, further reducing patch and undo data overhead.

Overlapping patches $p_1$ and $p_2$, with $p_2 \rightsquigarrow p_1$, may be merged *unless* future patches and dependencies might force the buffer cache to undo $p_2$, but not $p_1$. Reusing the reasoning developed for hard patches, we can carve out a class of patches that will never cause this problem: if $p_2$ is not the head of a block-level cycle containing $p_1$, then $p_2$ and $p_1$ can always be committed together.

To detect mergeable pairs, the Featherstitch implementation again uses a conservative heuristic that detects many pairs while limiting the cost of traversing dependency graphs. However, while the hard patch heuristic is both simple and effective, the heuristic for overlap merging has required some tuning to balance CPU expense and missed merge opportunities. The current version examines all dependency chains of uncommitted patches starting at $p_2$. It succeeds if no such chain matches $p_2 \rightsquigarrow x \rightsquigarrow p_1$ with $x$ on a different block, failing conservatively if any of the chains grows too long (more than 10 links) or there are too many chains. (It also simplifies the implementation to fail when $p_2$ overlaps with two or more soft patches that do not themselves overlap.) However, some chains cannot induce block-level cycles and are allowed regardless of how long they grow. Consider a chain $p_2 \rightsquigarrow x$ not containing $p_1$. If $p_1 \rightsquigarrow x$ as well, then since there are no circular dependencies, any continuation of the chain $p_2 \rightsquigarrow x$ will never encounter $p_1$. Our heuristic white-lists several such chains, including $p_2 \rightsquigarrow h$ where $h$ is a hard patch on $p_1$'s block. 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 $p_1$ to create a combined patch.

In our running example, overlap merging combines 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, 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 99.99%.

## 4.4 Ready Patch Lists

A different class of optimization addresses 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 $dep[P] \subseteq P \cup \boldsymbol{C}$. A naive implementation would guess a set $P$ and then traverse the dependency graph starting at $P$, looking for problematic dependencies. Patch merging limits the size of these traversals by reducing the number of patches. 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 bottleneck for cache sizes above 128 blocks (!).

Luckily, much of the information required for the buffer cache to choose a set $P$ can be precomputed. 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 a block has writable patches. To write a block $B$, the buffer cache initially populates the set $P$ with the contents of $B$'s 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 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 $p$ and $q$ actually commit.)

While the basic principle of this optimization is simple, its efficient implementation depends on several other optimizations, such as soft-to-hard patch merging, that preserve important dependency invariants. Although ready count maintenance makes some patch manipulations more expensive, ready lists save enough duplicate work in the buffer cache that the system as a whole is more efficient by multiple orders of magnitude.

## 4.5 Other Optimizations

Optimizations can only do so much with bad 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 $r$, $q$, and $p$ are adjacent on disk, the left-hand arrangement can be satisfied with two disk requests while the right-hand one will require four. Although neither arrangement is much harder to code, 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. Patches that change one independent field at a time generally give the best results. Featherstitch will merge these patches when appropriate, but if they cannot be merged, minimal patches tend to cause fewer patch reversions and give 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 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. Similarly, if *all* a file's directory entries will never exist on disk, the patches that free the file's blocks need not depend on the directory entries. Leaving out these dependencies can speed up the system by avoiding block-level cycles, such as those in Figure 4, and the rollbacks and double writes they cause. The Featherstitch ext2

module implements these optimizations, significantly reducing disk writes, patch allocations, and undo data required when files are created and deleted within a short time. Although the optimizations are file system specific, the file system implements them using general properties, namely, whether two patches successfully merge.

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. Allocating an indirect block in the middle of a range of file data blocks forces the data blocks to 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.

We often found it useful to examine patch dependency graphs visually. Featherstitch optionally logs patch operations to disk; a separate debugger inspects and generates graphs from these logs. Although the graphs could be daunting, they provided some evidence that patches work as we had hoped: performance problems could be analyzed by examining general dependency structures, and sometimes easily fixed by manipulating those structures.

# 5 Patchgroups

Currently, robust applications can enforce necessary write-before relationships, and thus ensure the consistency of on-disk data even after system crash, in only limited ways: they can force synchronous writes using `sync`, `fsync`, or `sync_file_range`, or they can assume particular file system implementation semantics, such as journaling. With the patch abstraction, however, a process might specify just dependencies; the storage system could use those dependencies to implement an appropriate ordering. This approach assumes little about file system implementation semantics, but unlike synchronous writes, the storage system can still buffer, combine, and reorder disk operations. This section describes *patchgroups*, an example API for extending patches to user space. Applications *engage* patchgroups to associate them with file system changes; dependencies are defined among groups. A parent process can set up a dependency structure that its child process will obey unknowingly. Patchgroups apply to any file system, including 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_depend(pg_t Q, pg_t P);   /* adds Q ⇝ P */
int pg_engage(pg_t P);        int  pg_disengage(pg_t P);
int pg_sync(pg_t P);          int  pg_close(pg_t P);
```

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 commit prior to any of those associated with $Q$. Engaging a patchgroup with `pg_engage` associates subsequent file system operations with that patchgroup. `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

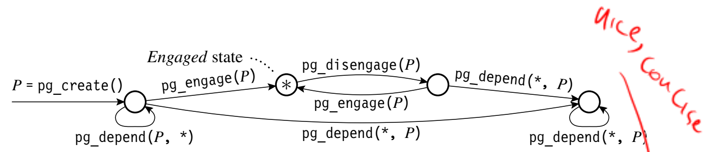



**Figure 5**: Patchgroup lifespan.

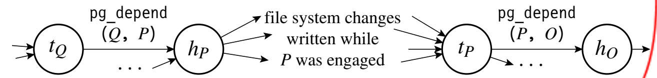



**Figure 6**: Patchgroup implementation (simplified). Empty patches $h_P$ and $t_P$ bracket file system patches created while patchgroup $P$ is engaged. `pg_depend` connects one patchgroup's $t$ patch to another's $h$.

be so well behaved, so the patchgroup API makes cycles unconstructable. Figure 5 shows when different patchgroup dependency operations are valid. As with patches themselves, all a patchgroup's direct dependencies are added first. After this, a patchgroup becomes engaged zero or more times; however, once a patchgroup $P$ gains a dependency via `pg_depend(*, P)`, it is sealed and can never be engaged again. This prevents applications from using patchgroups to hold dirty blocks in memory: $Q$ can depend on $P$ only once the system has seen the complete set of $P$'s changes.

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 the shell can connect pipe-oblivious programs into 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"
```

Inside the kernel, each patchgroup corresponds to a pair of containing empty patches, and each inter-patchgroup dependency corresponds to a dependency between the empty patches. All file system changes are inserted between all engaged patchgroups' empty patches. Figure 6 shows an example patch arrangement for two patchgroups. (The actual implementation uses additional empty patches for bookkeeping.)

These dependencies suffice to enforce patchgroups when using soft updates-like dependencies, but for journaling, some extra work is required. Since 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. Also, a metadata-only journal, which by default does not journal data blocks at all, pulls patchgroup-related data blocks into its journal, making it act like a full journal for those data blocks.

Patchgroups currently *augment* the underlying file system's consistency semantics, although a fuller implementation might let a patchgroup declare *reduced* consistency requirements as well.

## 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 consistency mechanisms, not to create new mechanisms. One effect of this choice is that versions of these applications could attain similar consistency guarantees by running

on a fully-journaled file system with a conventional API, although at least IMAP would require modification to do so. 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, showing that simple consistency requirements are simple to implement with patchgroups.

**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 this precaution may not protect a working copy operation against a crash. For example, the journal file is marked as complete by moving it from a temporary location to its 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, and the Subversion server (repository) library takes this approach, but developers deemed this approach too slow to be worthwhile at the client [25]. 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. This does not hold 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 express commit ordering requirements directly using patchgroups. The file rename property was replaced in two ways. Files created in a temporary location and then moved into their live location, such as directory status and journal files, now make the rename depend on the file data writes; but files only referenced by live files, such as updated file copies used by journal file entries, can live with a weaker ordering: the installation of referencing files is made to depend on the file data writes. The use of linearly ordered metadata updates was also replaced by patchgroup dependencies, and making the dependencies explicit let us reason about Subversion's actual order requirements, which are much less strict than linear ordering. For example, the updated file copies used by the journal may be committed in any order, and most journal playback operations may commit in any order. Only interacting operations, such as a file read and subsequent rename, require ordering.

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 safely committed to disk. 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).

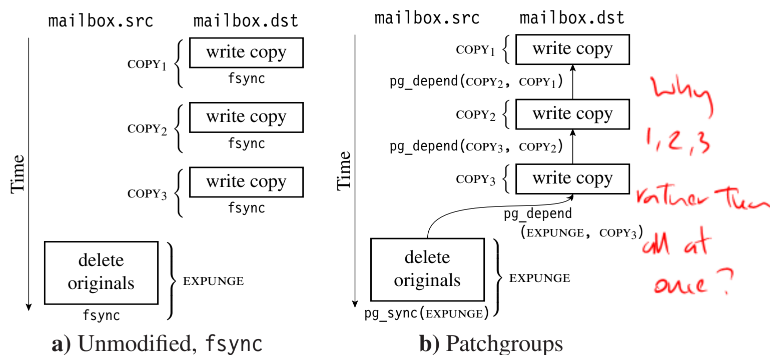



**Figure 7**: UW IMAP server, without and with patchgroups, moving three messages from `mailbox.src` to `mailbox.dst`.

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 and EXPUNGE commands use `pg_sync` to sync all preceding disk updates. This removes the requirement that COPY sync its destination mailbox: the client's CHECK or EXPUNGE 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 disk write ordering correctness and by performing disk writes more efficiently than synchronous writes. 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 modules that cooperate to implement file system functionality. Modules fall into three major categories. *Block device* (BD) modules are closest to the disk; they have a fairly conventional block device interface with interfaces such as "read block" and "flush." *Common file system* (CFS) modules are closest to the system call interface, and have an interface similar to VFS [15]. 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

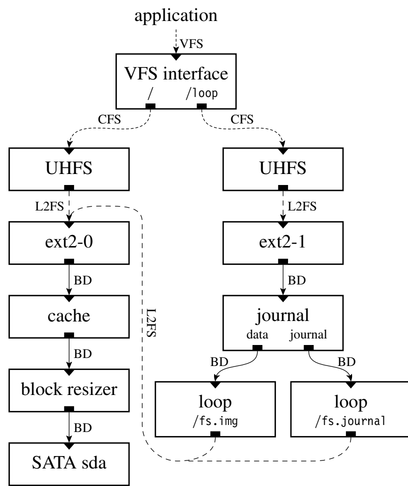



**Figure 8**: A running Featherstitch configuration. */* is a soft updated file system on an IDE drive; */loop* is an externally journaled file system on loop devices.

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. Our ext2 and UFS file system modules implement the L2FS interface.

Modules examine and modify dependencies via patches passed to them as arguments. For instance, every L2FS function that might modify the file system takes a `patch_t **p` argument. Before the function is called, `*p` is 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 [19]). 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 dependencies, they no longer need to concern themselves with file system consistency; 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 obey user-specified dependencies, such as patchgroups, by potentially shifting dependent patches into the current transaction. The journal format is similar to ext3's [34]: 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 any new dependencies do not introduce block-level cycles.

As in ext3, transactions are required to commit in sequence. 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, but ensures that in the event of a crash, the journal's committed transactions will be both contiguous and sequential.

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.

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.

## 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—are considered part of the "disk controller"; they can reorder block writes at will without violating dependencies, since those block writes will contain only in-flight patches. The buffer cache sees the complex consistency mechanisms that other modules define as nothing more than sets of dependencies among patches; it has no idea what consistency mechanisms 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 storage layer; unlike Linux's block device, consistency requirements on this block device are obeyed by the underlying file system. The loopback module forwards incoming dependencies to its underlying file system. As a result, the file system will honor those dependencies and preserve the loopback file system's consistency, even if it would normally provide no consistency guarantees for file data (for instance, if it used metadata-only journaling).

Figure 8 shows a complete, 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, maintaining dependencies across boundaries that might otherwise lose that information. Most systems cannot enforce consistency requirements through loopback devices this way—unfortunate, as file system images are becoming popular tools in conventional operating systems, used for example to implement encrypted home directories in Mac OS X. A simpler version of this configuration allows the journal module to store a file system's journal in a file on the file system itself, the configuration used in our evaluation.

## 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, a Featherstitch module graph replaces Linux's conventional file system layers. 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 tells the disk to 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 has several performance problems caused by incomplete Linux integration. For example, writing a block requires copying that block's data whether or not any patches were undone, and our buffer cache currently stores all blocks in permanently-mapped kernel memory, limiting its maximum size.

## 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 competitive with Linux, though using up to four times more CPU time; that Featherstitch file systems are consistent after system crashes; and that a patchgroup-enabled IMAP server outperforms the unmodified server on Featherstitch.

### 8.1 Methodology

All tests were run on a Dell Precision 380 with a 3.2 GHz Pentium 4 CPU (with hyperthreading disabled), 2 GB of RAM, and a Seagate ST3320620AS 320 GB 7200 RPM SATA2 disk. Tests use a 10 GB file system and the Linux 2.6.20.1 kernel with the Ubuntu v6.06.1 distribution. Because Featherstitch only uses permanently-mapped memory, we disable high memory for all configurations, limiting the computer to 912 MB of RAM. Only the PostMark benchmark performs slower due to this cache size limitation. 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 [14]. 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, 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 PostMark results for no optimizations and for just the hard patches optimization use a smaller maximum Featherstitch cache size, 80,000 blocks vs. 160,000 blocks, so that the benchmark does not run our machine out of memory. Optimization effectiveness is similar for journaling configurations.

Both optimizations work well alone, but their combination is particularly effective at reducing the amount of undo data—which,

| Optimization | # Patches | Undo data | System time |
|---|---|---|---|
| **Untar** | | | |
| 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** | | | |
| 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** | | | |
| None | 4,590,571 | 3,175.28 MB | 23.64 sec |
| Hard patches | 2,544,198 | 1,582.94 MB | 18.62 sec |
| Overlap merging | 550,442 | 1,590.27 MB | 12.88 sec |
| Both | 675,308 | 0.11 MB | 11.05 sec |
| **Andrew** | | | |
| None | 70,932 | 64.09 MB | 4.34 sec |
| Hard patches | 50,769 | 36.18 MB | 4.32 sec |
| Overlap merging | 12,449 | 27.90 MB | 4.20 sec |
| Both | 10,418 | 0.04 MB | 4.07 sec |

**Figure 9**: Effectiveness of Featherstitch optimizations.

| System | Untar | Delete | PostMark | Andrew |
|---|---|---|---|---|
| *Featherstitch ext2* | | | | |
| **soft updates** | **6.4 [1.3]** | **0.8 [0.1]** | **38.3 [10.3]** | **36.9 [4.1]** |
| **meta journal** | **5.8 [1.3]** | **1.4 [0.5]** | **48.3 [14.5]** | **36.7 [4.2]** |
| **full journal** | **11.5 [3.0]** | **1.4 [0.5]** | **82.8 [19.3]** | **36.8 [4.2]** |
| async | 4.1 [1.2] | 0.7 [0.2] | 37.3 [ 6.1] | 36.4 [4.0] |
| full journal | 10.4 [3.7] | 1.1 [0.5] | 74.8 [23.1] | 36.5 [4.2] |
| *Linux* | | | | |
| **ext3 writeback** | **16.6 [1.0]** | **4.5 [0.3]** | **38.2 [ 3.7]** | **36.8 [4.1]** |
| **ext3 full journal** | **12.8 [1.1]** | **4.6 [0.3]** | **69.6 [ 4.5]** | **38.2 [4.0]** |
| ext2 | 4.4 [0.7] | 4.6 [0.1] | 35.7 [ 1.9] | 36.9 [4.0] |
| ext3 full journal | 10.6 [1.1] | 4.4 [0.2] | 61.5 [ 4.5] | 37.2 [4.1] |

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

again, is pure overhead relative to conventional file systems. Undo data memory usage is reduced by at least 99.99%, the number of patches created is reduced by 85–99%, and system CPU time is reduced by up to 81%. These savings reduce Featherstitch memory overhead from 145–355% of the memory allocated for block data to 4–18% of that memory, a 95–97% reduction. For example, Featherstitch allocations are reduced from 3,321 MB to 74 MB for the PostMark benchmark, which sees 2,165 MB of block allocations.[2]

## 8.3 Benchmarks and Linux Comparison

We benchmark Featherstitch and Linux for all four benchmarks, comparing the effects of different consistency models and comparing patch-based with non-patch-based implementations. Specifically, we examine Linux ext2 in asynchronous mode; ext3 in writeback and full journal modes; and Featherstitch ext2 in asynchronous, soft updates, metadata journal, and full journal modes. All file systems were created with default configurations, and all journaled file systems used a 64 MB journal. Ext3 implements three different journaling modes, which provide different consistency guarantees. The strength of these guarantees is strictly ordered as "writeback < ordered < 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 the following 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 (FUA or a write-through cache) from unsafe results when comparing the systems. Although safe results for Featherstitch and Linux utilize different mechanisms (FUA vs. a write-through cache), we note that Featherstitch performs identically in these benchmarks when using either mechanism.

The results are listed in Figure 10; safe configurations are listed in bold. In general, Featherstitch performs comparably with Linux ext2/ext3 when providing similar durability guarantees. Linux ext2/ext3 sometimes outperforms Featherstitch (for the PostMark test in journaling modes), but more often Featherstitch outperforms Linux. There are several possible reasons, including slight differences in block allocation policy, but the main point is that Featherstitch's general mechanism for tracking dependencies does not significantly degrade total time. Unfortunately, Featherstitch can use up to four times more CPU time than Linux ext2 or ext3. (Featherstitch and Linux have similar system time results for the Andrew benchmark, perhaps because Andrew creates relatively few patches even in the unoptimized case.) 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 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 updates consistent: *fsck* reported, at most, 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 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 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

[2] Not all the remaining 74 MB is pure Featherstitch overhead; for example, our ext2 implementation contains an inode cache.

| Implementation | Time (sec) | # Writes |
|---|---|---|
| *Featherstitch ext2* | | |
| soft updates, fsync per operation | 65.2 [0.3] | 8,083 |
| full journal, fsync per operation | 52.3 [0.4] | 7,114 |
| soft updates, patchgroups | 28.0 [1.2] | 3,015 |
| full journal, patchgroups | 1.4 [0.4] | 32 |
| *Linux ext3* | | |
| full journal, fsync per operation | 19.9 [0.3] | 2,531 |
| full journal, fsync per durable operation | 1.3 [0.3] | 26 |

**Figure 11**: IMAP benchmark: move 1,000 messages. System CPU times shown in square brackets. Writes are in number of requests. All configurations are safe.

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 operation), Featherstitch with the patchgroup-enabled server (`pg_sync` on durable operations), and Linux and Featherstitch with the server modified to assume and take advantage of fully journaled file systems (changes are effectively committed in order, so sync only on durable operations). Only safe configurations are listed; unsafe configurations complete in about 1.5 seconds on either system. Featherstitch meta and full journal modes perform similarly; we report only the full journal mode. Linux ext3 writeback, ordered, and full journal modes also perform similarly; we again report only the full journal mode. Using an fsync per durable operation (CHECK and EXPUNGE) on a fully journaled file system performs similarly for Featherstitch and Linux; we report the results only for Linux full journal mode.

In all cases Featherstitch with patchgroups performs better than Featherstitch with fsync operations. Fully journaled Featherstitch with patchgroups performs at least as well as all other (safe and unsafe) Featherstitch and all Linux configurations, and is 11–13 times faster than safe Linux ext3 with the unmodified server. Soft updates dependencies are far slower than journaling for patchgroups: as the number of write requests indicates, each patchgroup on a soft updates file system requires multiple write requests, such as 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. The unmodified fsync-per-operation server generates dramatically more requests on Featherstitch with full journaling than Linux, possibly indicating a difference in fsync behavior. The last line of the table shows that synchronizing to disk once per durable operation with a fully journaled file system performs similarly to using patchgroups on a journaled file system. However, patchgroups have the advantage that they work equally safely, and efficiently, for other forms of journaling.

With the addition of patchgroups UW IMAP is able to perform mailbox modifications significantly more efficiently, while preserving mailbox modification safety. On a metadata or fully journaled file system, UW IMAP with patchgroups is 97% faster at moving 1,000 messages than the unmodified server achieves using fsync to ensure its write ordering requirements.

### 8.6 Summary

We find that our optimizations greatly reduce system overheads, including undo data and system CPU time; that Featherstitch has competitive performance on several benchmarks, 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 indicate 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 user space, 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.

For future work, we plan to improve performance further, particularly for system time; we have already improved performance by at least five orders of magnitude over the original implementation, but problems remain. The patch abstraction seems amenable to implementation elsewhere, such as over network file systems, and was designed to implement other consistency mechanisms like shadow paging. Finally, we would like to adapt patchgroups to 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 Linux kernel details, memory semantics, 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; by a Microsoft Research New Faculty Fellowship; and by an equipment grant from Intel. 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] Burnett, N. C. *Information and Control in File System Buffer Management*. PhD thesis, University of Wisconsin—Madison, July 2006.

[5] Cornell, B., P. A. Dinda, and F. E. Bustamante. Wayback: A user-level versioning file system for Linux. In *Proc. 2004 USENIX Annual Technical Conference, FREENIX Track*, pages 19–28, June 2004.

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

[7] Denehy, T. E., A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided resynchronization for software RAID. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 87–100, Dec. 2005.

[8] Gal, E. and S. Toledo. A transactional Flash file system for micro-controllers. In *Proc. 2005 USENIX Annual Technical Conference*, pages 89–104, Apr. 2005.

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

[11] Hitz, D., J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proc. USENIX Winter 1994 Technical Conference*, pages 235–246, Jan. 1994.

[12] Huang, H., W. Hung, and K. G. Shin. FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proc. 20th ACM Symposium on Operating Systems Principles*, pages 263–276, Oct. 2005.

[13] Kaashoek, M. F., D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proc. 16th ACM Symposium on Operating Systems Principles*, pages 52–65, Oct. 1997.

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

[15] Kleiman, S. R. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proc. USENIX Summer 1986 Technical Conference*, pages 238–247, 1986.

[16] Liskov, B. and R. Rodrigues. Transactional file systems can be fast. In *Proc. 11th ACM SIGOPS European Workshop*, Sept. 2004.

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

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

[19] McKusick, M. K., W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, Aug. 1984.

[20] Muniswamy-Reddy, K.-K., C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *Proc. 3rd USENIX Conference on File and Storage Technologies (FAST '04)*, pages 115–128, Mar. 2004.

[21] Nightingale, E. B., P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles*, pages 191–205, Oct. 2005.

[22] Nightingale, E. B., K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–14, Nov. 2006.

[23] Quinlan, S. and S. Dorward. Venti: a new approach to archival storage. In *Proc. 1st USENIX Conference on File and Storage Technologies (FAST '02)*, pages 89–101, Jan. 2003.

[24] Rosenthal, D. S. H. Evolving the Vnode interface. In *Proc. USENIX Summer 1990 Technical Conference*, pages 107–118, Jan. 1990.

[25] Rowe, M. Re: wc atomic rename safety on non-ext3 file systems. Subversion developer mailing list, Mar. 5 2007. `http://svn.haxx.se/dev/archive-2007-03/0064.shtml` (retrieved August 2007).

[26] Seltzer, M. I., 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 *Proc. 2000 USENIX Annual Technical Conference*, pages 71–84, June 2000.

[27] Sivathanu, G., S. Sundararaman, and E. Zadok. Type-safe disks. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 15–28, Nov. 2006.

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

[29] Sivathanu, M., A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Jha. A logic of file systems. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 1–15, Dec. 2005.

[30] Sivathanu, M., L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Database-aware semantically-smart storage. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 239–252, Dec. 2005.

[31] Skinner, G. C. and T. K. Wong. "Stacking" Vnodes: A progress report. In *Proc. USENIX Summer 1993 Technical Conference*, pages 161–174, June 1993.

[32] Soules, C. A. N., G. R. Goodson, J. D. Strunk, and G. R. Ganger. Metadata efficiency in versioning file systems. In *Proc. 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, pages 43–58, Mar. 2003.

[33] Ts'o, T. Re: [evals] ext3 vs reiser with quotas, Dec. 19 2004. `http://linuxmafia.com/faq/Filesystems/reiserfs.html` (retrieved August 2007).

[34] Tweedie, S. Journaling the Linux ext2fs filesystem. In *Proc. 4th Annual LinuxExpo*, 1998.

[35] Vilayannur, M., P. Nath, and A. Sivasubramaniam. Providing tunable consistency for a parallel file store. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 17–30, Dec. 2005.

[36] Waychison, M. Re: fallocate support for bitmap-based files. linux-ext4 mailing list, June 29 2007. `http://www.mail-archive.com/linux-ext4@vger.kernel.org/msg02382.html` (retrieved August 2007).

[37] Wright, C. P. *Extending ACID Semantics to the File System via ptrace*. PhD thesis, Stony Brook University, May 2006.

[38] Wright, C. P., M. C. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proc. 2003 USENIX Annual Technical Conference*, pages 197–210, June 2003.

[39] Wright, C. P., 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.

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

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

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

[43] Zadok, E., I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. In *Proc. 1999 USENIX Annual Technical Conference*, pages 57–70, June 1999.