Generalized File System Dependencies

r3305:3306M—Paper #169—14 pages

Abstract

File systems ensure that their data is kept consistent through careful write ordering, where one block cannot be written until another is safely committed to stable storage. Previous file systems have enforced write orderings in system-dependent ways, either with rules specialized for each file system structure [10] or with a journal, which enforces a particular consistency protocol. We present a general *patch* abstraction that can represent any write ordering in a file system agnostic manner. A patch-based file system implementation expresses dependencies among writes, but not how those dependencies should be enforced. File system modules can examine, preserve, and modify write orderings. Generalized file system dependencies are naturally exportable to user level, allowing applications to specify their own consistency protocols for the file system to follow.

We present the patch abstraction, describe a number of important optimizations for patch-based file systems, and present a Linux kernel implementation of a storage subsystem that uses patches to enforce consistency. Our ext2 prototype is competitive with FreeBSD soft updates and allows several novel configurations, such as ext2 with soft updates or correct dependency enforcement within a loopback file system.

1 Introduction

This paper aims to evaluate whether a simple, unified abstraction that represents all modifications to stable storage, including *dependencies* among modifications, could support an entire file system, where modifications are common and cache sizes are large. The answer is a qualified yes.

As file system functionality increases, file system correctness is increasingly a focus of research [8, 28]. File systems must maintain dependency relationships between written blocks to preserve file system correctness. This is difficult enough for file system implementers [18, 33], but further complicated by file system extensions and special disk interfaces [6, 20, 23, 27, 29, 31, 35]. Although stackable file systems [12, 24, 37, 38] place extensions over an existing file system responsible for maintaining consistency, some extensions—for instance, creating a file containing metadata information about a directory—can introduce ordering requirements difficult to express at the VFS layer. This problem exists for applications as well. The expensive `fsync` and `sync` system calls are the only tools available for enforcing file consistency. Robust applications, including databases, mail servers, and source code management tools, either accept the performance penalty of these system calls or rely on specialized raw-disk interfaces.

Proposed systems for improving file system integrity and consistency differ mainly in the kind of consistency they aim to impose, ranging from metadata consistency to full data journaling and even full ACID transactions [9, 16]. However, different extensions within a file system, or different applications over the file system, may require different types of consistency semantics, and performance suffers when lower layers are unnecessarily denied the opportunity to reorder writes. A better choice might be an abstraction that could express many consistency models.

This work develops a new fundamental data type called a *patch*, which represents both a change to disk data and any *dependencies* between that change and other changes. Patches were inspired by the dependency abstraction from BSD's soft updates [10], but whereas the soft updates implementation involves many structures specific to the UFS file system [18], patches are fully general, simply specifying how a range of data should be changed. This lets file system implementations and extensions examine and modify dependency structures independent of the file system's layout. This generality comes at a cost, but the cost can be partially mitigated. Real file system implementations achieve consistency and write ordering relationships using a variety of file system specific optimizations, and we found that a naive implementation scaled terribly poorly, spending far more space and time on dependency manipulation than conventional systems. We therefore developed a variety of dynamic optimizations that which significantly reduce the memory and CPU overhead associated with patches. We describe patches abstractly, state their behavior and safety properties, and reason about the correctness of our optimizations. Finally, we demonstrate a prototype file system implementation using patches called Dodder. For an untar-Linux benchmark that writes 253 MB of data, our optimizations can reduce the number of patches created by half and the amount of rollback data memory required by 99.73%.

The Dodder file system implementation is decomposed entirely into pluggable modules that manipulate patches, hopefully making the system as a whole more configurable, extensible, and easier to understand. Any file system module can generate patches; other modules can examine them and modify them when required. Patch dependency requirements are obeyed by all other file system layers, allowing them to be passed through layers such as loopback block devices. As a result, the loosely-coupled modules that make up a file system implementation can cooperate to implement strong and often complex consistency guarantees, even though each individual module does a relatively small part of the work.

Patches can implement many consistency mechanisms, including soft updates and journaling. Our file system modules impose soft updates-style patch requirements by default; we have also written a prototype journal module. Additionally, we give user-level applications controlled access to patches, allowing applications to impose their own limited consistency policies. Modifying an IMAP mail server to use this interface requires only localized changes; the resulting server follows IMAP's consistency requirements while writing fewer blocks to disk. Our prototype implementation is not yet as fast as we would like, but it already runs several benchmarks faster than FreeBSD on the same hardware.

Our contributions include the generalized patch design, the optimizations we have developed for dealing with patches, the formal model of patches, the module interfaces in our Dodder prototype, several of the individual Dodder modules like the journal, and the patchgroup interface that exports patches to user space.

2 Related Work

Consistency Soft updates [10] significantly lowers the overhead required to provide file system consistency. By carefully ordering writes to disk, soft updates avoids the need for synchronous writes to disk or duplicate writes to a journal. Soft updates also guarantees a strong level of consistency after a crash, enough so that the system can avoid time-consuming file system consistency checks using a utility like *fsck*. A comparable approach to protecting the integrity of the file system [25] is to write upcoming operations to a journal first. The content and the layout of the journal vary in each implementation, but in all cases, the system can use the journal to play out (or roll back) the operations that did not complete as a result of a crash. Thus, *fsck* can be avoided by consulting the journal when recovering from a crash. Section 5 explains journaling with patches.

External synchrony [22] builds on journaling to automatically provide strict file system operation ordering for applications, without requiring them to block on each write. It combines operations into a journal, but tracks the activity of the calling processes after returning control to them from the file system. If a process later performs some *user-visible* operation like printing text to the screen or sending network traffic, the journal transaction containing the changes is forced to commit before the process can continue. External synchrony depends inherently on dependency tracking; dependencies among processes with outstanding data are tracked to ensure that uncommitted output never reaches a user. Xsyncfs depends on a particular file system consistency methodology, namely journaling; Dodder patch dependencies could be a natural implementation strategy for Xsyncfs dependencies, allowing them to apply to any file system. Similar patch-like dependencies are used to improve network file system performance in BlueFS [21].

Customizable application-level consistency protocols have previously been considered in the context of distributed, parallel file systems by CAPFS [34] and Echo [17]. 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. It also provides a mechanism for applications to extend the partial order, thus reducing the server's flexibility to choose how to write the data or make it available to other clients. 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. However, this is also true with a local file system, and as a result applications must use expensive interfaces like `fsync()` when they require specific consistency guarantees. Dodder brings this sort of customizable consistency to all applications, not just those using specialized distributed file systems. A similar application interface to our patchgroups is explored in Section 4 of Burnett's thesis [5].

Stackable File Systems Stackable module software for file systems continues to attract active research [11, 12, 24, 30, 35, 36, 37, 38]. Previous systems like FiST [37] or GEOM [2] generally focus on an individual portion of the system and thus restrict both what a module can do and how modules can be arranged. FiST, for instance, does not provide a way to deal with structures on the disk directly – it provides only "wrapper" functionality around existing file systems. GEOM, on the other hand, deals only with the block device layer, and has no way to work with the file systems stored on those block devices. Neither has a formal way of specifying or honoring complex write-ordering information, which is what patches in Dodder provide. We imagine that systems like these could be adapted to work with patches, giving the benefits of both ideas.

Applications A variety of extensions to file systems and disk interfaces have been proposed in recent work, like the FS2 Free Space File System [13], encrypting file systems like NCryptfs [35], and type-safe disks [26]. The Dodder module system may provide an interesting platform for implementations of these ideas.

3 Patch Model

Every change to stable storage in a Dodder system is represented by a *patch*. This section describes the basic patch abstraction using a semi-formal notation that will be useful for analyzing our optimizations later. Although the patch idea would apply to any stable medium, to network file systems, or to multiple disks, we use terms like "the disk" and "the disk controller" throughout to simplify our terminology.

Each patch p encapsulates four important pieces of information: its *block*, its *state*, a set of *direct dependencies*, and some *rollback data*.

Patch p's **block**, written $p.block$, is the unit of disk data to which p applies. The disk controller is assumed to execute writes a block at a time (although not necessarily atomically; see below). A file system change that affects n blocks must be represented using at least n patches, since a patch by definition can touch only one block.

Each patch is in one of three **states**: *in memory*; *in flight* to the disk controller, but not yet committed to disk; and *on disk*. The intermediate in-flight state is necessary because operating system software loses control of a patch upon writing its block to disk. p's state is written $p.state \in \{mem, flight, disk\}$. The operating system changes a patch's state from *mem* to *flight* by writing its block to the disk controller. Some time later—after possible controller-level disk scheduling, transit over the system bus, and a period in the disk's cache—the disk writes the block to stable storage and reports success. When the processor receives this notification, it changes the patch's state to *disk*. The sets *Mem*, *Flight*, and *Disk* are defined to contain all patches with the given state, and the sets $Mem[b]$, $Flight[b]$, and $Disk[b]$ are defined to contain all patches with the given state *on the given block b*.

Patch p's **direct dependencies**, written $p.ddeps$, is a set of other patches on which p "depends". That is, every patch in $p.ddeps$ should be committed to disk either before, or at the same time as, p itself. Dependencies are set by the file system based on the consistency semantics it wants to ensure. For example, a file system with asynchronous writes (and no durability guarantees) might write all patches with $p.ddeps = \varnothing$; a file system implementing soft updates would arrange the dependencies accordingly; and a file system implementing journaling would write a different set of dependencies, where the journal commit record would depend on the journal data and the writes to the main body of the file system would in turn depend on the commit record. In summary, an upper file system layer defines an initial set of dependencies; the rest of Dodder generally preserves this initial set, but can modify it as necessary to change dependency semantics; and finally, the buffer cache obeys the constraints thus defined.

We say *p directly depends on q*, and write $p \to q$, when $q \in p.ddeps$. We say *p depends on q*, and write $p \rightsquigarrow q$, when there exists a dependency chain from p to q: that is, either $p \to q$ or, for some patch x, $p \rightsquigarrow x \to q$. Patch p's set of *dependencies* is written $Dep[p] = \{q \mid p \rightsquigarrow q\}$; this is of course a superset of its direct dependencies. Given a *set* of patches P, we write $Dep[P]$ to mean the set's combined dependencies $\bigcup_{p \in P} Dep[p]$.

Rollback data. Soft updates-like consistency orderings may require that the buffer cache *not* write one or more patches on some block. In particular, a series of file system operations may create dependencies that enforce a circular order among blocks, even though the dependencies themselves do not form a cycle [10]. This is problematic since blocks with circular dependencies can never be written: no block can be written first since each block depends on another. For this reason, each patch carries *rollback* information that gives the previous version of the data altered by the patch. If a patch p is not written with its containing block, the system rolls back the patch, which swaps the new data and the previous version. Once the block is written, the system will roll the patch forward and, when allowed, write the block again, this time including the patch. Rollback information adds greatly to memory and CPU utilization, but it can often be optimized away, as we show below.

Our patch notation is summarized in Figure 1.

$p.block = b$	p's block
$p.state$	p's state, $\in \{mem, flight, disk\}$
$p.ddeps$	p's direct dependencies: patches that must go to disk before p
Mem	all in-memory patches: $\{p \mid p.state = mem\}$
$Mem[b]$	$\{p \mid p.state = mem$ and $p.block = b\}$
$Flight, Disk$	similarly for in-flight and on-disk patches
$p \to q$	p directly depends on q: $q \in p.ddeps$
$p \rightsquigarrow q$	p depends on q: either $p \to q$ or there exists a patch x so that $p \rightsquigarrow x \to q$
$Dep[p]$	p's dependencies: $\{q \mid p \rightsquigarrow q\}$

Figure 1: Patch notation.

3.1 Obeying Dependencies

Dodder must ensure the **disk safety property**, which states that the dependencies of all patches on disk are also on disk:

$$Dep[Disk] \subseteq Disk.$$

Thus, no matter when the system crashes, the disk is consistent in terms of dependencies. The file system's job is to set up dependencies so that the disk safety property implies file system correctness.

However, Dodder can only control when patches are handed to the disk controller, not when they are written to disk. Disk controller behavior is encapsulated in the following atomic action:

```
Commit block:
    Pick some block $b$ with $Flight[b] \neq \varnothing$.
    For each $p \in Flight[b]$, set $p.state \leftarrow disk$.
```

Since the controller can write blocks in any order, Dodder must ensure that in-flight blocks are mutually independent. This is precisely stated by the **in-flight safety property:**

$$\text{For all blocks } b,\ Dep[Flight[b]] \subseteq Flight[b] \cup Disk.$$

This ensures that for any $b' \neq b$, $Dep[Flight[b]] \cap Dep[Flight[b']] \subseteq Disk$; the disk controller can safely write in-flight blocks in any order and still preserve disk safety.

The buffer cache must thus behave according to the following atomic action:

```
Write block:
    Pick some block $b$ with $Mem[b] \neq \varnothing$.
    Pick some $P \subseteq Mem[b]$ with $Dep[P] \subseteq P \cup Disk$.
    For each $p \in P$, set $p.state \leftarrow flight$.
    For each $p \in Mem[b] - P$, set $p.ddeps \leftarrow p.ddeps \cup P$.
```

Since the disk controller can write in-flight blocks in any order, at most one version of a block can be in flight at any time. Thus, the last step above forces any rolled-back patches to wait in memory at least until the in-flight version of the block is written. Likewise, any patch created on a block with in-flight patches must depend on those patches. (These dependencies are implicit in our implementation.) The buffer cache is also expected to write all blocks eventually, a liveness property.

This model does not completely define the disk's behavior on system crash, in particular with respect to in-flight blocks. Soft updates inherently assumes that blocks are written *atomically*, except in the case of catastrophic media error: if the disk fails while a block b is in flight, then b's value on recovery must equal either the old value or the new value. Most journal designs do not rely on this assumption, and can recover properly even if in-flight blocks are corrupted—for instance, because the memory holding the new value of the block lost its coherence before the disk stopped writing [22]. However, some disks may actually provide an atomicity guarantee, for instance by using non-volatile memory to store blocks before they make it onto disk. The Dodder core makes no assumptions about block atomicity, instead relying on software above it to implement a consistency protocol that makes sense for the given disk.

The buffer cache can't handle cross-block dependency cycles since it must write data in units of blocks. Thus, if $p \rightsquigarrow q \rightsquigarrow p$, then p and q must be on the same block; otherwise, neither p nor q could ever get written. Whenever upper layers might generate cross-block dependency cycles, intervening layers must break those cycles somehow before they reach the cache. For instance, a journal layer might detect dependency cycles and put the corresponding patches into a single transaction, but the *implementation* of that transaction could not contain cross-block cycles. In practice, the Dodder implementation currently disallows any dependency cycles, including cycles within a single block; this simplifies many graph traversals.

3.2 Dodder Interface and Empty Patches

Dodder modules create patches with functions like `patch_-create_byte`. Arguments to these functions include the relevant block ($p.block$), any direct dependencies ($p.ddeps$), and the new data represented by the patch. Most patches specify the new data as a contiguous byte range, including an offset into the block and the patch length in bytes. The rollback data for very small patches (4 bytes or less) is stored in the patch structure itself; for larger patches, rollback data is stored in separately allocated memory. As a useful special case, patches that flip one or more bits in a word (for instance, in a free block bitmap) are represented specially.

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

For each block b, Dodder maintains a list of all patches with $p.block = b$. However, the implementation does not keep track of on-disk patches; when patch p commits, Dodder destroys p and removes all dependencies $q \rightarrow p$. The resulting behavior follows our model. For each patch p, Dodder maintains lists of its direct dependencies (all q where $p \rightarrow q$) and its direct "reverse dependencies" (that is, all q where $q \rightarrow p$).

Dodder also supports *empty* patches, which have no associated data or block. Empty patches are useful for representing sets of dependencies and for preventing other patches from being written. For example, while a journal transaction is being populated in a journaling file system, many changes to the main body of the disk should depend on a journal commit record that has not yet been created. Dodder therefore makes these patches depend on a "managed" 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. Once the commit record is written, the empty patch is automatically "written" as well, which allows the main file system changes to follow. Some aspects of our model would have to change to handle empty patches; for example, the in-flight safety property would change to:

$$Dep[Flight[b]] \subseteq Flight[b] \cup Disk \cup Empty.$$

We call a patch *true* if it is not empty.

Empty patches both aid and hurt performance. On the one hand, empty patches can represent quadratic sets of dependencies with a linear number of edges: if all m patches in P must depend on all n patches in Q, one could add an empty patch x and $m+n$ direct dependencies $p_i \rightarrow x \rightarrow q_j$. On the other hand, some functions may have to traverse trees of empty patches to determine true dependencies.

4 Patch Optimizations

A naive implementation of this model has all the properties we want except good performance. File system modules can define their own consistency protocols, other modules can modify them, applications can affect consistency orderings, and it is all relatively easy thanks to the patch abstraction. Unfortunately, this abstraction is extremely expensive.

Consider an unoptimized version of Dodder running an ext2 file system with soft updates-like dependencies. Figure 2a shows the patches generated when an application appends 16 kB of data to an existing empty file. This requires allocating four new 4 kB blocks, writing data to them, attaching them to the file's inode, changing the inode's file size and modification time, and updating the "group descriptor" and superblock to account for the allocated blocks. The operation is broken into four one-block appends, each of which updates the inode; note, for example, how overlap dependencies force each modification of the inode's size to depend on the previous one. Eight blocks are written during the operation. Dodder, however, represents the operation with 23 patches and roughly 33000 (!) bytes of rollback data. The patches slow down the buffer cache system by making graph traversals more expensive. The 4096 bytes of rollback data per patch stored for “clear data” and “write data” patches, which initialize and write data blocks, is particularly painful since in this example, data blocks *never* need to be rolled back!

This section shows how the actual Dodder system uses generic patch properties and dependency analysis to reduce the 23 patches and 33000 bytes of rollback data in Figure 2a to the 8 patches and 0 bytes of rollback data in Figure 2d. These optimizations apply transparently to any Dodder file system.

4.1 Hard Patches

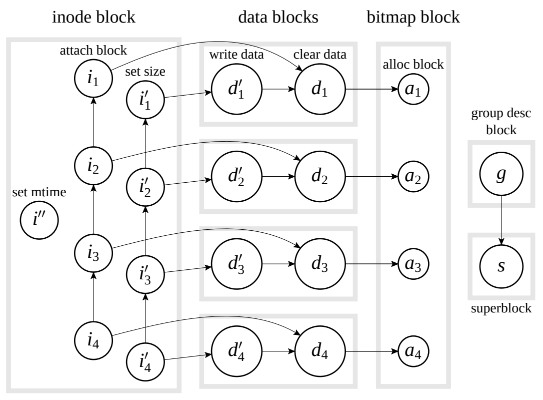
The first optimization reduces the space overhead of rollback data by detecting when it will never be necessary. When a patch p is created, Dodder conservatively detects whether p will ever need to be rolled back. If not, p is a *hard patch*: its data can be applied directly to the in-memory copy of the block without saving rollback data for the previous version.

Only patches in a block-level dependency cycle might need to be rolled back. However, some of the patches in a cycle may be hard; only the crucial patch that creates the cycle must be soft, for that is the patch that must be rolled back to create a valid write order. Thus, patch p may be hard if it will *never* be the head of a block-level cycle: that is, if there will never exist true patches q and p' with $p \rightsquigarrow q \rightsquigarrow p'$ and $p.block = p'.block \neq q.block$. In other words, if the only cycle between $p.block$ and $q.block$ is $p \rightarrow q \rightarrow p'$, then q and p' may be hard, but p may not. As a result, the set of all hard patches in memory will never contain a block-level cycle, and the system can always find a valid order to write hard patches to disk. This assumes that the dependency structure correctly represents dependencies among patches on the same block. Specifically, all hard patches on a block depend on each other (since they can’t be written independently), and any soft patch on a block depends on that block’s hard patches, if any (since it cannot be written without them).

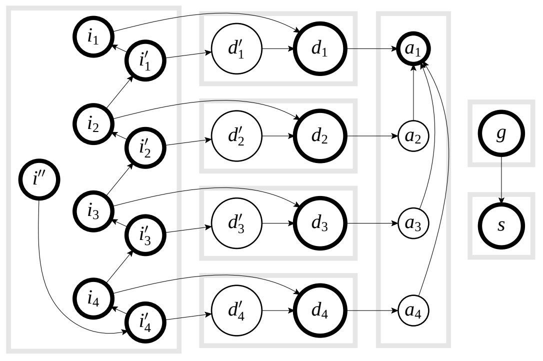
When patch p is created, Dodder must decide whether any future patch or dependency could create a block-level cycle with p at the head. This decision is simplified by an API property: *All of a patch’s direct dependencies are specified at creation time*; no new dependencies $p \rightarrow q$ may be introduced after p is created.[1] Since every patch follows this rule, all possible block-level cycles $p \rightsquigarrow q \rightsquigarrow p'$ are visible at creation time.

Traversing the dependency graph to check for a block-level cycle proved to be quite expensive. Dodder thus implements a conservative approximation: patch p is created as hard if *no* other block depends on the in-memory version of $p.block$.

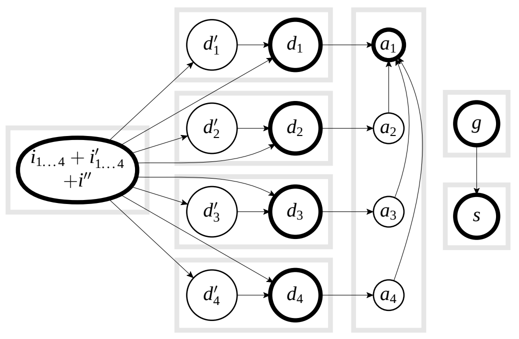
[1]The actual rule is somewhat more flexible: modules may add new direct dependencies if they guarantee that those dependencies don’t introduce any new block-level cycles. As one example, if no patch depends on some empty patch p, then adding a new $p \rightarrow q$ dependency can’t introduce a cycle. Additionally, modules can specify at creation time that a patch is not allowed to become hard, because dependencies might be added to it later.



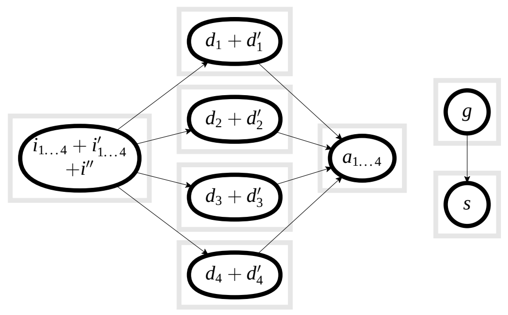
a) Naive implementation



b) With hard patches . . .



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



d) . . . plus overlap merging

Figure 2: Patches required to append 4 blocks to an existing file, without and with optimization. Hard patches are shown with heavy borders.

That is, for all dependencies $q \rightsquigarrow p'$ with $p' \in Mem[p.block]$, either $q.block = p.block$ or q is an empty patch. If no other block depends on p's block, then p can never initiate a block-level cycle no matter its dependencies. Determining this property is less expensive than a full graph traversal, and works well in practice.

Dodder further ensures that the dependency structure correctly represents dependencies on the same block through overlap dependencies: since hard patches are considered to cover the entire block, every succeeding patch will overlap at least one hard patch, and Dodder will automatically add a dependency. (Some cases are handled by other optimizations.)

The buffer cache's "write block" behavior must account for hard patches, as it *must* write any hard patches that exist on a block. Let $Hard[b]$ be the set of hard patches on block b. Then to write block b, the buffer cache must choose some $P \subseteq Mem[b]$ with

$$Dep[P] \subseteq P \cup Disk \text{ and } Hard[b] \cap Mem \subseteq P.$$

If no such P exists, then the cache must write a different block.

Applying hard patch rules to our example makes 14 of the 23 patches hard (Figure 2b), reducing the rollback data required by slightly more than half.

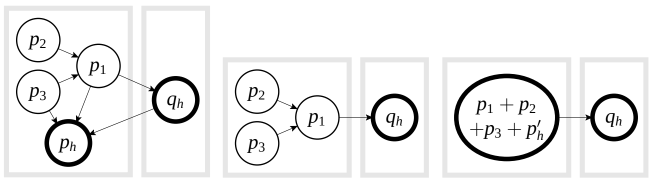
4.2 Hard Patch Merging

File operations such as block allocations, inode updates, and directory updates create many distinct patches. Keeping track of these patches and their dependencies requires significant amounts of memory and CPU time. Dodder therefore *merges* patches when possible, drastically reducing the 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, Dodder updates the data and dependencies so as to merge the new patch into the existing one.

The first type of merging is trivial to explain: since all of a block's hard patches must be written at the same time, there is no need to maintain more than one hard patch per block. All other hard patches can safely be merged into the first, and in fact Dodder ensures that each block contains at most one hard patch.

Assume some module attempts to create a patch p on a block that already has a hard patch h, and Dodder determines that p may be hard. Then the implementation merges p into h at creation time by applying p's data to the block and setting $h.ddeps \leftarrow h.ddeps \cup p.ddeps$. Although this changes h's direct dependency set, that change cannot introduce any new block-level cycles or p would not have been hard in the first place. The existing hard patch h is returned to the caller.

Unfortunately, the merge can create *intra*-block cycles. If some empty patch x exists with $p \rightsquigarrow x \rightsquigarrow h$, then after the merge $h \rightsquigarrow x \rightsquigarrow h$. The merge algorithm uses a graph traversal to prune these cyclic dependencies as it merges p's dependencies into h.



a) Initial state **b)** After p_h reaches disk **c)** Soft-to-hard conversion

Figure 3: Soft-to-hard conversion.

Hard patch merging is able to eliminate 8 of the patches in our running example, as shown in Figure 2c.

Soft-to-Hard Conversion A new hard patch might also be created on a block that has existing *soft* patches in memory. For example, Figure 3a shows a block with both hard and soft patches: p_1 must be soft since it initiates a block-level cycle $p_1 \rightsquigarrow q_h \rightsquigarrow p_h$, and similarly for p_2 and p_3. To write these blocks, the buffer cache must roll back p_1, p_2, and p_3 and write p_h on its own. Once p_h reaches disk, the patches in memory reach the state in Figure 3b. Now that p_h has moved to *Disk*, it is irrelevant for cycle detection, and in fact there are no block-level dependency cycles in the diagram: it would be safe to transform p_1, p_2, and p_3 into hard patches. However, Dodder delays this and performs the conversion only later when adding a new hard patch to the p block.

Consider a new hard patch p_h that is added to a block that contains some soft patch p. Since p_h overlaps p, Dodder adds a direct dependency $p_h \rightarrow p$. Since p_h could be hard, we know there are no block-level dependency cycles $p_h \rightsquigarrow q \rightsquigarrow p'$. But as a result, we know that there are no block-level dependency cycles with the existing *soft* patch as head, since such a cycle would imply a p_h-headed cycle $p_h \rightarrow p \rightsquigarrow q \rightsquigarrow p'$. Thus, p can be transformed into a hard patch and then merged into p_h using hard patch merging. Figure 3c shows the resulting configuration when a new hard patch p'_h is added.

4.3 Overlap Merging

The final type of merging merges soft patches with other patches, hard or soft, when they overlap. Bitmap blocks and inodes accumulate many nearby and overlapping patches as data is appended to or truncated from a file. Figure 2 shows how even data blocks can collect overlapping dependencies: actual data writes d'_j overlap, and therefore depend on, block initialization writes d_j, but cannot be made hard since when they are created another block (the inode) already depends on the data block. Luckily, simple reasoning can identify many mergeable pairs, further reducing the number of patches and the amount of rollback data required.

Two overlapping patches p_1 and p_2, with $p_1 \rightsquigarrow p_2$, may be merged iff it would always be possible to write them at the same time. Here we may reuse the reasoning developed for hard patches above: it is always possible to write these patches simultaneously if, assuming that p_2 were hard, p_1 could also be made hard—that is, if p_1 will never be the head of a block-level cycle terminating at p_2. The same properties that simplified the creation of hard patches also help us check this property: that is, if no block-level cycle $p_1 \rightsquigarrow x \rightsquigarrow p_2$ exists when p_1 is created, then no such block-level cycle will ever exist.

As with hard patch creation, the Dodder implementation checks a simpler property that requires less graph traversal. It checks that every path starting at p_1 fits at least one of the following cases:

- $p_1 \longrightarrow q$, where $q \notin \mathit{Mem}$.
- $p_1 \longrightarrow p_2$.
- $p_1 \longrightarrow h$, where h is the hard patch on $p_1.\mathit{block}$.
- $p_1 \longrightarrow q$, where q depends on no other patch.
- $p_1 \longrightarrow q \longrightarrow r$, where r depends on no other patch, q depends on nothing but r, and $r.\mathit{block} \neq p_1.\mathit{block}$.

If all paths fit, then there are no block-level cycles from p_1 to p_2, p_1 and p_2 can have the same lifetime, and p_1 can be merged into p_2 where they overlap. (This may require growing p_2 to cover p_1's data range.) We initially began with a simpler check, namely that p_1 *only* depended on p_2, but extending the check to length-2 non-branching paths allowed much more merging (further extensions did not). It also simplifies the implementation somewhat to limit overlap merging to the case when neither p_1 nor p_2 overlaps with any other patch.

In our running example, overlap merging is able to combine all remaining soft patches with their hard counterparts, reducing the number of patches to the minimum of 8 and the amount of rollback data to the minimum of 0. In our experiments, we observe that hard patches and our patch merging optimizations reduce the amount of memory allocated for rollback data in soft updates orderings by up to 99.73%.

4.4 Ready Patch Lists

Our final important optimization precomputes much of the information required for the buffer cache to choose a set of patches to write. For each patch p we maintain counts that specify whether or not p is immediately ready to write to disk.

The buffer cache's main task is to choose sets of patches P that satisfy the in-flight safety property $\mathit{Dep}[P] \subseteq P \cup \mathit{Disk}$. Redundant patch graph traversals to calculate valid P sets would severely limit cache size scalability. Dodder therefore explicitly tracks, for each patch, how many of its direct dependencies are in memory or in flight. These counts are updated as patches are added to the system, and as the system receives notifications that patches have reached 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. (Empty patches add a slight complication, since an empty patch "reaches the disk" as soon as all its dependencies reach the disk. The counting algorithms propagate this state change automatically.)

Some patches in P may depend on other patches in P; however, since Dodder eliminates all dependency cycles, if P is nonempty then at least one patch in P depends only on other blocks. When that patch p becomes ready, it will be added to P, and Dodder subtracts its count from other patches *on the same block*. This may make more patches on the block ready, which will in turn be added to P, and so forth.

To write a block, the buffer cache thus iterates through the block's ready list, sending patches to the target block device, until the list is empty. The buffer cache can also immediately tell whether a block *could possibly* be written by checking whether its ready list is empty. On-line maintenance of the ready counts adds some cost to several patch manipulations, but since it saves so much duplicate work in the buffer cache the resulting system is more efficient by multiple orders of magnitude—and in particular, CPU time no longer scales superlinearly with the size of the cache.

4.5 Other Optimizations

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

$$\begin{tikzcd}
& p \arrow[dl] \arrow[d] \arrow[dr] & \\
q & r & s
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
& p \arrow[dl] & \\
q \arrow[r] & r \arrow[r] & s
\end{tikzcd}$$

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

Care must also be taken to avoid unnecessary *implicit* dependencies, and in particular overlap dependencies. For instance, inode blocks contain multiple inodes, and changes to two inodes should generally be independent; a similar statement holds for directories and even sometimes for different fields in a summary block like the superblock. The best results can be obtained from *minimal* patches that change one independent field at a time; Dodder will merge these patches when appropriate, but if they cannot be merged, minimal patches will lead to fewer rollbacks and more flexibility in write ordering.

Finally, the buffer cache and a few other modules perform better in the common case that the patches on a block are listed in order of creation time. This improves their performance from $O(n^2)$ to $O(n)$ in the number of patches.

5 CONSISTENCY MODELS

Soft updates, journaling, and other consistency models correspond to different patch arrangements. Dodder's current file system modules generate dependencies corresponding to soft updates orderings by default [10]. Adding an independent

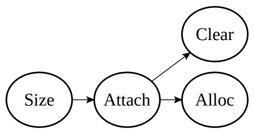


Figure 4: Soft updates patches for extending a UFS inode by one block. Attaching the block pointer to the inode depends on initializing the block (Clear) and updating the free block map (Alloc). Updating the inode's size depends on writing the block pointer.

journal module rearranges the patches to provide journaling semantics.

Soft updates Soft updates enforces orderings between disk writes that maintain a relatively consistent file system state at all times, speeding reboot by avoiding *fsck*. Not all invariants can be preserved, so soft updates relaxes the least critical: a crash or sudden reboot can leak blocks or other disk structures, but will never create more serious inconsistencies. In the BSD UFS implementation of soft updates, each UFS operation is represented in memory by a structure encapsulating the disk changes necessary to implement that operation. As a result, many specialized data structures represent the different possible file system operations. These structures, their relationships to one another, and their uses for tracking and enforcing dependencies are quite complex [18].

Dodder, in contrast, represents all dependencies by patches and relies on general optimizations to reduce memory usage. The resulting system is not as optimized as the BSD implementation, but much simpler to specify and easier for other modules to manipulate. The patches that make up a file system operation are connected to specify the order in which the changes must be written to disk. For instance, most file systems require at least two patches to remove block from a file, one (p) to clear the block reference from the file's block list and one (q) to mark the block as free; adding the $q \rightarrow p$ dependency straightforwardly implements soft updates semantics. Figure 4 shows another example, the patches generated in UFS when a file is extended by one block.[2]

Journaling In a journaling file system, changes to disk structures are grouped into *transactions* that commit atomically. Any change in a transaction is first copied into an on-disk journal. A single *commit record* block is written to the journal once the transaction's changes are stably committed there. This record commits the transaction itself, allowing its changes to be written into the main body of the file system in any order. If the system crashes, the journal information is used to recover the main file system from its possibly-incomplete state. Once all the transaction's changes are written, a *completion record* is written to the journal to mark the transaction as complete; that portion of the journal may now be reused.

The Dodder journal module sits below a regular file system and transforms incoming patches into journal transactions.

[2]The dependencies differ from the ext2 dependencies for a similar operation in Figure 2 because of a difference in file system semantics.

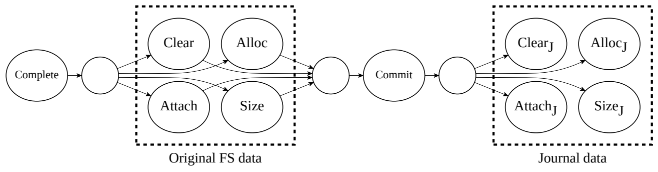


Figure 5: Journal patch subgraph for the change in Figure 4. The unlabeled circles are empty patches.

Data patches 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. The dependencies among the original patches are removed since the journal itself provides consistency for each high-level file system operation. The journal format is similar to ext3's [33]: a transaction contains a list of block numbers, the data to be written to those blocks, and finally a single commit record. Figure 5 shows how the journal module transforms the patches in Figure 4 into journaling semantics. Note that the journal must modify existing patches' direct dependencies; this is allowed since the new dependencies will never introduce a block-level cycle.

Our journal module prototype ignores incoming dependencies and enforces transactions based on high-level file system operations. It is thus uniformly enforces a particular journal semantics—namely, that each file system operation happens atomically—regardless of the semantics specified by the file system above it. It should be possible, however, to extend the journal module to obey dependencies.

Asynchronous writes Finally, we also wrote a trivial module that removes all dependencies from incoming patches, allowing the buffer cache to write blocks in any order. This implements similar semantics to existing file systems like ext2 in asynchronous write mode.

6 Modules

A complete Dodder configuration is composed of many modules, making it a finer-grained variant of a stackable file system. There are has three major types of modules. Closest to the disk are block device (BD) modules, which have a fairly conventional block device interface with interfaces such as "read block" and "flush". Closest to the system call interface are *common file system* (CFS) modules, which have an interface similar to VFS [15]. Dodder also supports an intermediate interface between BD and CFS. This *low-level file system* (L2FS) interface helps divide file system implementations into code common across block-structured file systems and code specific to a given file system layout. The L2FS interface has functions to allocate blocks, add blocks to files, allocate file names, and other file system micro-operations. A module implementing the L2FS interface defines how bits are laid out on the disk, but doesn't have to know how to combine the micro-operations into larger, more familiar file system operations. A generic CFS-to-L2FS module called UHFS ("universal high-level file system") decomposes the larger file write, read, append, and other standard operations into L2FS micro-operations. File system extensions like those often implemented by stackable file systems would generally use the CFS interface; for example, we wrote a simple CFS module that provides case-insensitive access to a case-sensitive file system. File system implementations, such as our ext2 and UFS implementations, use the L2FS interface.

Patches are explicitly part of the L2FS interface. Every L2FS function that might modify the file system takes a `patch_t **p` argument. When the function is called, the patch `*p` is set to the patch, if any, on which the modification should depend; when the function returns, the patch `*p` must be set to depend on the modification itself. (Empty patches allow this interface to generalize to multiple dependencies.) For example, this function is called to append a block to an L2FS inode (which is called "`fdesc_t`"):

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

This design lets L2FS modules examine and modify the dependency structure.

6.1 Write-Back Cache

The Dodder buffer cache module is called the *write-back cache*. Its job is twofold: first, it caches blocks in memory, and second, it ensures that modifications are written to stable storage in an order that preserves the in-flight safety property. Modules "below" the write-back cache—that is, between its output interface and the disk—can reorder block writes at will without violating dependency orderings. There can be many write-back caches in a configuration at once (for instance, one for each block device). To a write-back cache, the complex consistency protocols that other modules want to enforce are nothing more than sets of dependencies among patches; it has no idea what consistency protocols it is implementing, if any.

Our prototype write-back 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 write-back cache finds a block with ready patches, all other patches on that block are rolled back, the resulting block data is copied and sent to the disk driver, the ready patches are marked in-flight, and the rolled-back patches are rolled forward again. The block itself is also marked in-flight, ensuring that the cache will not try to write it again until the current version commits.

As a special case to aid disk performance, when the write-back cache finds it can write 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

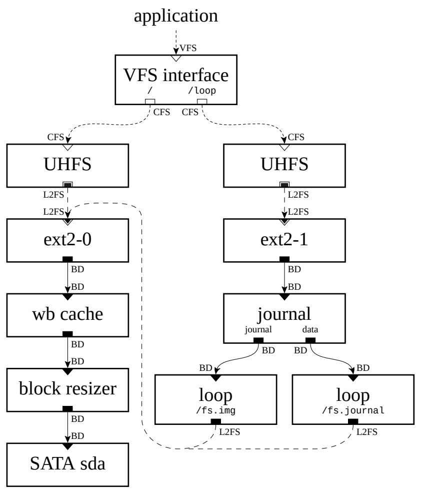


Figure 6: A running Dodder configuration. */* is a soft updated file system on an IDE drive; */loop* is an externally journaled file system on loop devices.

in Linux's elevator scheduler. Nevertheless, our profiling results indicate significant 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 roll back patches unnecessarily when a different order would have required fewer writes.

6.2 Loopback Module

The loopback module is a BD module that uses a file in an L2FS module as its underlying data store. It is very similar to the device of the same name in Linux, but with one critical difference: it is aware of patches. The loopback module preserves its file system's dependencies and forwards them to its underlying data store. As a result, the data store will honor those dependencies and preserve the loopback file system's consistency, even if the data store would normally provide no guarantees for consistency of file system data (e.g., it used metadata-only journaling).

Figure 6 shows an example configuration using the loopback module. A file system image is mounted with an external journal, both of which are loopback block devices stored on the root file system (which uses soft updates). The journaled file system's ordering requirements are sent through the loopback module as patches, allowing dependency information to be maintained across boundaries that might otherwise lose that information. In contrast, without patches and the ability to forward patches through loopback devices, BSD cannot express soft updates' consistency requirements through loopback devices. The modules in Figure 6 are a complete and working Dodder configuration, and although the use of a loopback device is somewhat contrived in the example, they are increasingly being used in conventional operating systems. For instance, Mac OS X uses them in order to allow users to encrypt their home directories.

6.3 ext2 and UFS

Dodder 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 soft updates-like dependencies; 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. We verified that file systems generated by our modules are considered correct by their reference implementations on FreeBSD and Linux by mounting and running *fsck* on Dodder-generated disk images.

Both modules are implemented at the L2FS interface. The modules create patches for all their changes to the disk and connect them to form subgraphs that enforce the soft updates rules [10] as applied to each file system. The UHFS module is also aware of soft updates order when necessary; when it implements a single operation using multiple L2FS calls, it hooks the resulting patches up in the correct order. Unlike FreeBSD's soft updates implementation, once the dependencies have been hooked up, the ext2 and UFS modules no longer need to concern themselves with their patches, as the block device subsystem tracks and enforces the dependency orderings.

7 PATCHGROUPS

Robust applications ensure their on-disk data remains in a sane state even after a system crash by requiring their changes be committed in a specified order. Existing systems can ensure such orders in two ways: applications may tell the file system *how* to implement the ordering by requiring that all changes on a particular file (or the whole disk) be committed with `(f)sync`, or applications can simply require particular file system implementation semantics, such as journaling. The patch abstraction, in contrast, can let a process specify *what* ordering is required and leave the file system to implement that ordering in the most appropriate way.

This section describes *patchgroups*, which extend patches to user space and let applications express dependencies among file system operations. Compared to a dictated order, this lets the file system optimize commit orderings, but is still relatively independent of the underlying file system. For example, patchgroups even allow an application to specify dependencies among raw block device writes.

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

7.1 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 pg1, pg_t pg2);
int pg_engage(pg_t pg);     int pg_disengage(pg_t pg);
int pg_sync(pg_t pg);       int pg_close(pg_t pg);
```

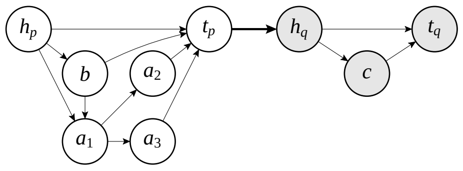


Figure 7: Patches corresponding to two patchgroups, p and q. The h and t patches are created by the patchgroup module; the heavy dependency between t_p and h_q was added by `pg depend(p, q)`. Each of a_i, b, and c corresponds to a different file system change.

Each process has its own set of patchgroups. The call `pg_depend(p, q)` makes patchgroup `p` depend on patchgroup `q`: every change associated with `p` will be delayed until all changes associated with `q` have committed. Patchgroups can be *engaged* or *disengaged*; all changes made by a process are associated with that process's engaged patchgroups. `pg_sync` forces an immediate write of a patchgroup to disk. Whereas Dodder modules are presumed to not create cyclic dependencies, the kernel cannot safely trust user applications to be so well behaved, so the patchgroup API restricts dependency manipulation so that cycles are inconstructible. For example, `pg_depend(p, q)` returns an error if `p` has ever been engaged.

Patchgroups and file descriptors are managed similarly; e.g. they are copied across `fork()` and preserved across `exec()`. This allows existing programs to interact with patchgroups without knowing it, in the same way that programs do not have to know about pipes for the shell to connect them in a pipeline. For example, a `depend` program could apply patchgroups to unmodified applications by setting up the patchgroups before calling `exec`. The following command line would ensure that `in` is not removed until all changes in the preceding `sort` have committed to disk:

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

Patchgroup support is implemented by an L2FS module that lives above a file system module like ext2. Each patchgroup corresponds to a pair of empty patches, and inter-patchgroup dependencies affect these patches. When an patchgroup is engaged, the L2FS module ensures that all file system changes are inserted between these patches. Figure 7 shows an example patch arrangement for two patchgroups.

7.2 Case Studies

We tested the patchgroup interface by adding patchgroup support to three applications: the gzip compression utility, the Subversion version control system, and the uw-imapd IMAP mail server daemon.

Gzip Our first test showed that trivial consistency requirements were trivial to add. We updated gzip to use patchgroups that ensure the input file's removal is not committed before the output file's data is written, so that a crash does not lose both files. The update adds 10 lines of code to gzip.

Subversion The Subversion version control system's client [3] manipulates a local *working copy* of a possibly-remote source 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. Even with this precaution, however, a working copy operation may not be protected against an operating system or hardware crash. For example, the journal file is committed when it is moved from a temporary to its official location, but the file system might commit this rename to stable storage *before* completely writing the journal file's data. A system crash in between the complete commit of these two operations and the subsequent journal replay could corrupt the working copy.

The working copy library could ensure a safe commit ordering by syncing files as necessary; the Subversion server (repository) library takes this approach. However, as is common for non-server applications, the Subversion developers deemed this approach too slow to be worthwhile at the client [32]. Instead, the working copy library takes advantage of two file system ordering properties ensured by ordered data and full data journaling, such as is implemented by ext3. First, all preceding writes to a file's data are committed before the file is renamed, and second, metadata updates are committed in their system call order. In addition to being obtuse, file renaming does not behave as required on many systems other than ext3 in ordered or full data mode; for example, neither NTFS nor BSD UFS with soft updates provide this property. And non-journaled systems, such as BSD UFS with soft updates, do not provide the metadata ordering property. On these systems, the journal file's installation may precede earlier file installations, leaving the working copy in an inconsistent state.

We updated the Subversion working copy library to use patchgroups to express commit ordering requirements without relying on properties of the underlying file system implementation. For example, Figure 8 shows the patchgroups created to update a file with conflicts. The use of the file rename property is replaced in two ways. First, files created in a temporary location and then moved into their official location (e.g. directory status and journal files) now make the rename depend on the file data writes. Second, files only referenced by official files (e.g. updated file copies used by journal file entries) can live with a weaker ordering; the data for these files need only precede the installation of the referencing file. The use of linearly ordered metadata updates is also replaced by patchgroup dependencies. Although the original operations assumed a linear ordering of metadata updates, the actual order requirements are considerably less strict. For example, the updated file copies used by the journal may be committed independently of each other. A second example, most journal playback operations may commit independently of each other; only interacting operations, such as a file read and subsequent rename, need ordering.

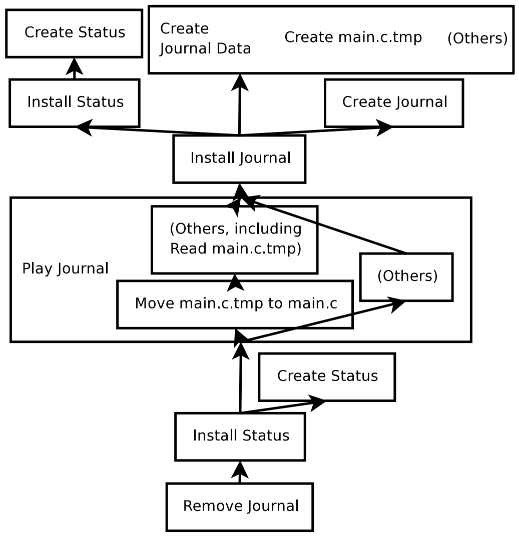


Figure 8: Patchgroups to update a file in a Subversion working copy.

Once we understood Subversion's requirements, it took a day to add the 220 lines of code that enforce safety for conflicted updates (out of 25000 in the working copy library).

UW IMAP Finally, we updated the University of Washington's IMAP mail server [4] to ensure mail updates are committed to disk in safe and efficient orderings. The Internet Message Access Protocol (IMAP) [7] 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).

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

These changes to UW IMAP simplify it in two ways: ensuring change ordering correctness and making efficient usage of disk updates. As each command's changes now depend on the preceding command's changes, it is no longer required that all code 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 9.4.

8 Implementation

The Dodder prototype implementation runs as a Linux 2.6 kernel module. It interfaces with the Linux kernel at the VFS layer and the generic block layer. In between is a Dodder module graph that replaces Linux's conventional file system layers and buffer cache. We bypass all Linux caches as the simplest way to ensure that the Dodder buffer cache obeys all necessary dependency orderings. A small kernel patch informs Dodder of process fork and exit events as required to update per-process patchgroup state.

Dodder modules interact with Linux's generic block 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 Dodder when the disk controller reports request completion; for writes, this sets the corresponding patch states to *disk*. The disk safety property requires that the disk controller wait to report completion until a write has reached stable storage. Most drives instead report completion when a write has reached the drive's volatile cache. Ensuring the stronger property could be quite expensive, requiring frequent barriers or setting the drive cache to write-through mode; either choice seems to prevent older drives from reordering requests. The solution is a combination of SCSI tagged command queuing (TCQ) or SATA native command queuing (NCQ) with either a write-through cache or "forced unit access" (FUA). TCQ and NCQ allow a drive to independently report completion for multiple outstanding requests, and FUA is a per-request flag that says the disk should report completion only after the request reaches stable storage. Recent SATA drives handle NCQ plus write-through caching or FUA exactly as we would want: the drive appears to reorder write requests, improving performance drastically relative to older drives, but reports completion only when data reaches the disk. We use a patched version of the Linux 2.6.20 kernel with good support for NCQ and FUA, and a recent SATA2 drive.

Future work will require hardening several aspects of our Linux integration. Integrating patch and dependency support into Linux's native buffer cache would allow applications to use memory-mapped I/O on Dodder files and reduce Dodder's memory usage. We also observe that read delay increases notably when there are many writes. We suspect that write requests, which for Dodder take far longer than Linux expects because the drive must delay completion notification, are using all available command slots; one or more slots should probably be reserved for reads.

Optimization	**# patches**	**Rollback data**	**System time**
	Untar test		
None	537,989	458.64 MB	4.42 sec
Hard patches	537,131	205.68 MB	4.53 sec
Overlap merging	253,839	255.45 MB	3.58 sec
Both optimizations	276,829	1.24 MB	3.58 sec
	Rm test		
None	193,062	3.17 MB	1.14 sec
Hard patches	172,351	3.15 MB	1.08 sec
Overlap merging	87,692	1.83 MB	0.85 sec
Both optimizations	54,076	0.64 MB	0.75 sec

Figure 9: Effectiveness of Dodder optimizations. The combination of hard patches and overlap merging removes almost half the patches and 99.73% of the rollback data required to untar Linux.

9 Evaluation

Our evaluation shows that a Dodder patch-based file system layer has overall performance that is within reason, even though it is slower than Linux by a nontrivial amount. We also demonstrate the effectiveness of patchgroups on some of our sample applications. All tests were run on a Dell Precision 380 with a 3.2GHz Pentium 4 CPU, 2GB of RAM, and a Seagate ST3320620AS 320GB 7200RPM SATA2 disk, using 10GB file systems. All tests use a 10 GB file system. Our tests use Linux 2.6.20.1 with ext2 and ext3, Dodder with ext2 with soft updates, and FreeBSD 6.0 with UFS in soft updates mode, all created with default configurations.

Two tests are repeated throughout this section: first, we untar the Linux 2.6.15 source code from `linux-2.6.15.tar` (218 MB), which is already decompressed and cached in RAM. Second, we delete the resulting source tree, after unmounting and remounting the file system.

9.1 Optimization Benefits

Dodder benefits greatly from the optimizations discussed in Section 4. The total number of patches created, amount of rollback data allocated, and system CPU time used are shown for the untar and rm tests in Figure 9, with all combinations of using hard patches and overlap merging. While both optimizations work well by themselves, the combination of the two is particularly effective at reducing the amount of rollback data. The system time column shows time spent executing code in kernel space during the untar or remove; this is mostly Dodder overhead. Linux's system time numbers during a comparable test are much lower. Profiling numbers indicate optimization opportunities in some of Dodder's auxiliary libraries, such as its hash table, as well as more significant optimization opportunities we hope to address in future work.

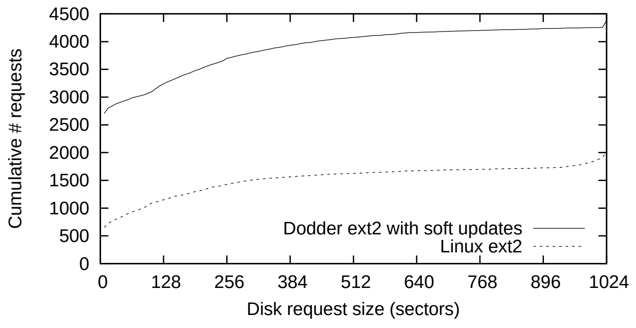


Figure 10: CDF of disk write request sizes for Linux ext2 and Dodder ext2 during the untar test. Linux totals 2021 requests to the disk, while Dodder totals 4382 requests.

System	Untar (sec)	Delete (sec)
Dodder (SU)	18.39	9.47
Dodder (async)	3.88	6.94
FreeBSD (SU)	23.22	15.95
FreeBSD (async)	10.09	3.25
Linux (ext2)	5.27	4.43
Linux (ext3)	13.92	4.47

Figure 11: Untar and delete times.

9.2 Microbenchmarks

Our first microbenchmark measures the number of blocks written by Linux ext2 and ext3 and Dodder ext2 with soft updates during the untar test. Linux ext2 writes 523,248 blocks; Linux ext3 writes 523,192; Dodder writes 540,472.

We also tested that Dodder's soft updates ext2 writes a similar number of blocks as FreeBSD's soft updates UFS. In one test, we create 100 small files in a directory and measure the number of blocks written to disk. Dodder writes 122 blocks, while FreeBSD writes 135 blocks.

Finally, Figure 10 plots a CDF of the sizes of disk write requests during the untar test. Here we can see one likely contributing factor to the speed difference between Linux and Dodder: Dodder makes over twice as many requests to the disk. We suspect that this is due to a suboptimal cache eviction algorithm, as described above.

9.3 Macrobenchmarks

Times to untar and remove our Linux tarball are listed in Figure 11 for several systems, including time to fully sync the changes to disk.

Dodder is about 21% faster than FreeBSD using soft updates for writes, and about 41% faster for deletions.[3] When compared to Linux ext3, Dodder is about 32% slower for writes, but many times slower for deletions. (A major cause of the slow delete performance is writing updates to blocks which have been freed, which we have a plan to optimize.) This demonstrates that the overhead of using patches in Dodder is not entirely unreasonable, but has room for significant improvement.

[3]Interestingly, FreeBSD 5.4 performs better than version 6.0 at these tests, particularly the delete test, but Dodder is still faster on the untar test.

System	Time (sec)
Dodder	46.6
Linux (ext2)	16.9
Linux (ext3)	19.6
FreeBSD	27.0

Figure 12: PostMark times.

System	# writes (blocks)
Dodder (with patchgroups)	919
Dodder (w/o patchgroups)	1537
FreeBSD 5.4	2200

Figure 13: Number of block writes to move 100 IMAP messages.

We also tested the correctness of our soft updates implementation by implementing a module that crashes the operating system, without giving it a chance to synchronize its buffers, at a random time during the untar test. Removing dependencies from our ext2 file system represents similar guarantees as an asynchronous file system; after crashing, fsync reported that the file system contained many references to inodes that had been deleted: the file system was corrupt. With our soft updates dependencies, the file system was mostly consistent: fsync reported that inode reference counts were higher than the correct values (an expected discrepancy after a soft updates crash).

We next evaluate overall performance using the PostMark benchmark, designed to simulate the small file workloads seen on email and netnews servers [14]. We use PostMark 1.5, configured to use 4kB reads and writes and to create 500 files ranging in size from 500 B to 2 MB; perform 500 transactions consisting of file reads, writes, creates, and deletes; and finally delete its files. PostMark is a single process, single threaded program that performs these file system operations as quickly as possible. Figure 12 shows the times to run PostMark and then synchronize its results to disk.

Dodder's PostMark performance, compared to Linux and FreeBSD, falls into the middle of §9.3's untar and delete benchmarks. As PostMark combines file creation and deletion, its results follow for similar reasons as the untar and delete benchmarks'.

9.4 Patchgroups

We implemented a preliminary assessment of the patchgroup-enabled UW IMAP mail server (§7.2) by comparing the number of block writes that Dodder makes relative to FreeBSD to move 100 emails. The test selects the source mailbox (with 100 messages, sized 2kB each), creates the new mailbox, copies each of the 100 messages to the new mailbox, marks each source message for deletion, expunges the marked messages, requests a check, and logs out. We compare using soft updates; Figure 13 shows the results.

10 CONCLUSION

Dodder provides a new way for file system implementations to formalize write ordering requirements, allowing file system software to be divided into modules that cooperate loosely to implement strong consistency guarantees. We also explored several optimizations to the patch abstraction that significantly improve performance. Patches simplify the design of consistency protocols like journaling and soft updates by separating the specification and enforcement of write ordering requirements. Additionally, patches can be extended outside the file system implementation into userspace, allowing applications to specify in a limited way what their specific consistency requirements are, providing the file system implementation more freedom to reorder writes without violating the application's needs. Extending the patch abstraction into user space may be able to improve the efficiency of applications with specific write ordering requirements.

REFERENCES

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

[2] *GEOM – modular disk I/O request transformation framework*. `http://www.freebsd.org/cgi/man.cgi?query=geom&sektion=4`.

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

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

[5] N. C. Burnett. *Information and Control in File System Buffer Management*. PhD thesis, 2006.

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

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

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

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

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

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

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

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

[14] J. Katcher. PostMark: A new file system benchmark. TR0322, Network Appliance, 1997.

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

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

[17] T. Mann, A. Birrell, A. Hisgen, C. Jerian, and G. Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems (TOCS)*, 12(2):123–164, 1994. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/176575.176577.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[32] Subversion developers. personal communication, 2007.

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

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

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

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

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

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