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/

Supported by the NSF, Microsoft, and Intel.

Featherstitch Summary

- A new architecture for constructing file systems
- The generalized dependency abstraction
  - Simplifies consistency code within file systems
  - Applications can define consistency requirements for file systems to enforce

File System Consistency

- Want: don’t lose file system data after a crash
- Solution: keep file system consistent after every write
  - Disks do not provide atomic, multi-block writes

- Example: journaling

Log Journal Transaction ← Commit Journal Transaction ← Update File System Contents

- Enforce *write-before relationships*

File System Consistency Issues

- Durability features vs. performance
  - Journaling, ACID transactions, WAFL, soft updates
  - Each file system picks one tradeoff
  - Applications get that tradeoff plus sync
- Why no extensible consistency?
  - Difficult to implement
  - Caches complicate write-before relations
  - Correctness is critical

> "Personally, it took me about 5 years to thoroughly understand soft updates and I haven't met anyone other than the authors who claimed to understand it well enough to implement it." – Valerie Henson

> FreeBSD and NetBSD have each recently attempted to add journaling to UFS.
> Each declared failure.

The Problem

Can we develop a simple, general mechanism
for implementing *any* consistency model?

Yes! With the *patch* abstraction in Featherstitch:

- File systems specify low-level write-before requirements
- The buffer cache commits disk changes, obeying their order requirements

Featherstitch Contributions

- The *patch* and *patchgroup* abstractions
  - Write-before relations become explicit and file system agnostic
- Featherstitch
  - *Replaces* Linux's file system and buffer cache layer
  - ext2, UFS implementations
  - Journaling, WAFL, and soft updates,
    implemented using just patch arrangements
- Patch optimizations make patches practical

Patches

Problem

Patches for file systems

Patches for applications

Patch optimizations

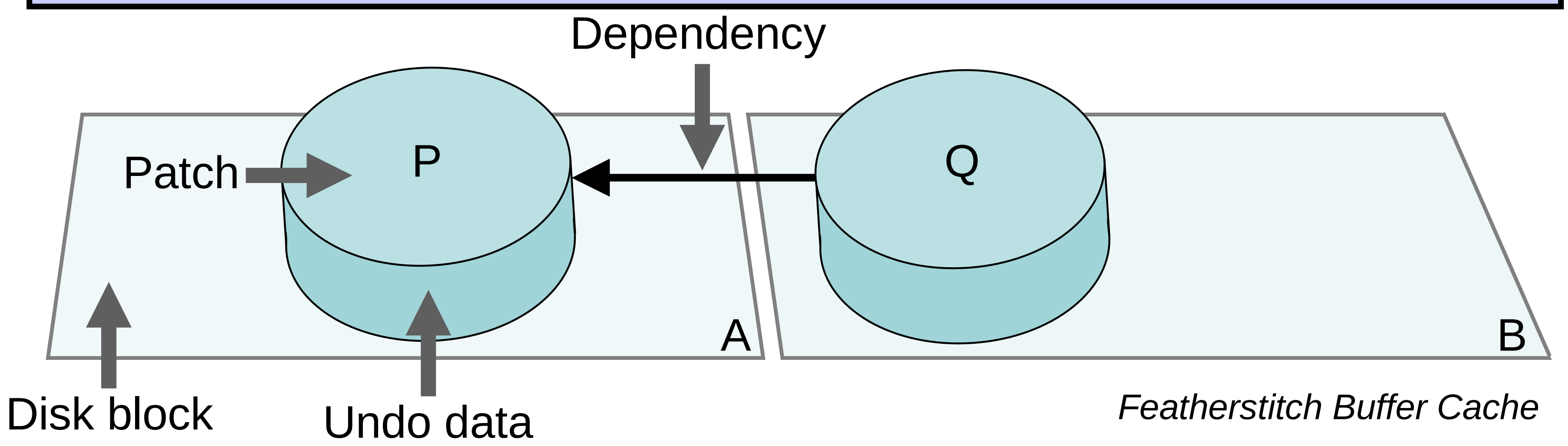
Evaluation

Patch Model

A patch represents:

- a disk data change
- any dependencies on other disk data changes

patch_create(block* **block**, int **offset**, int **length**, char* **data**, patch* **dep**)



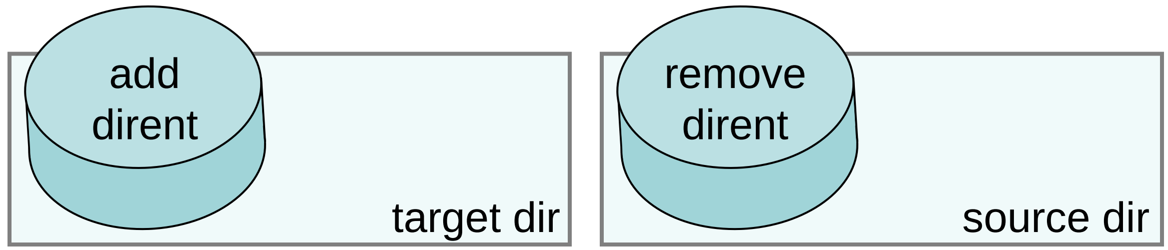
Benefits:

- separate write-before specification and enforcement
- explicit write-before relationships

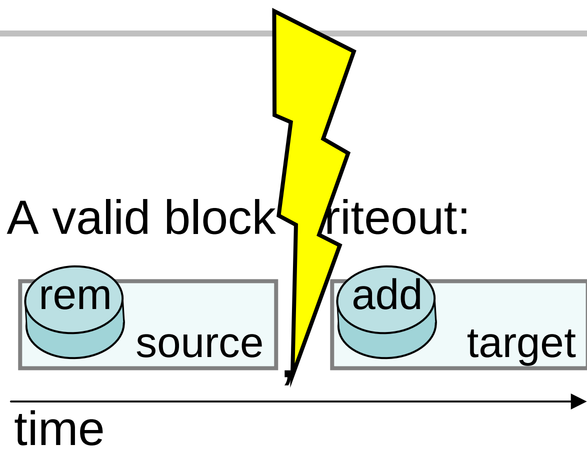
Base Consistency Models

- Fast
  - Asynchronous
- Consistent
  - Soft updates
  - Journaling
- Extended
  - WAFL
  - Consistency in file system images
- All implemented in Featherstitch

Patch Example: Asynchronous rename()

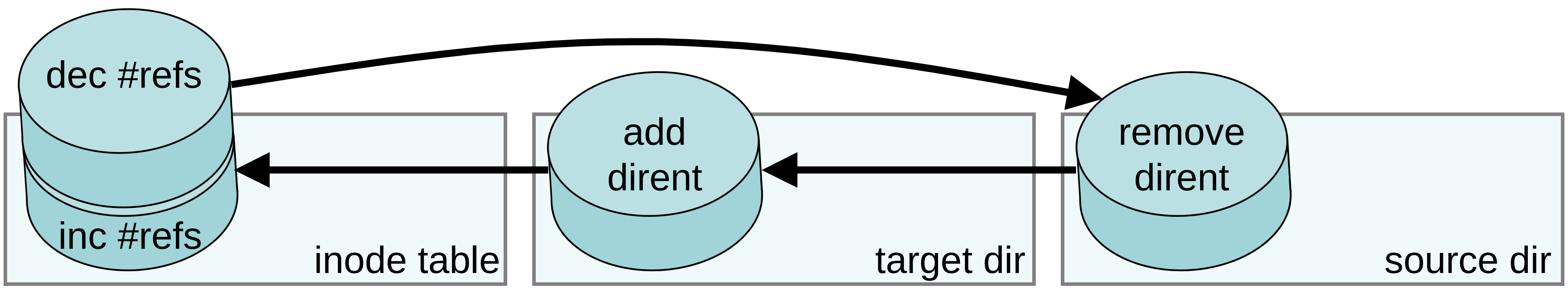


A valid block writeout:

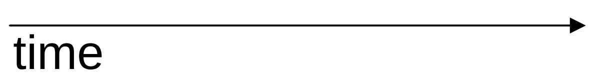


File lost.

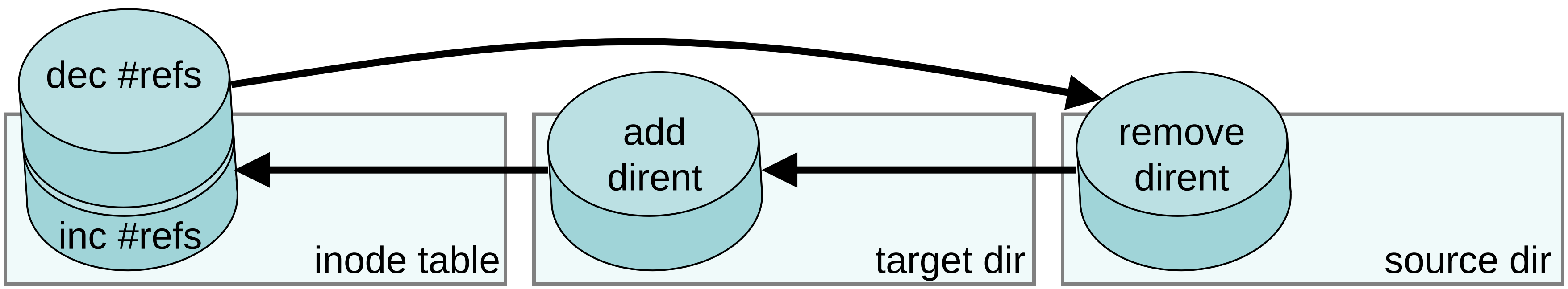
Patch Example: rename() With Soft Updates



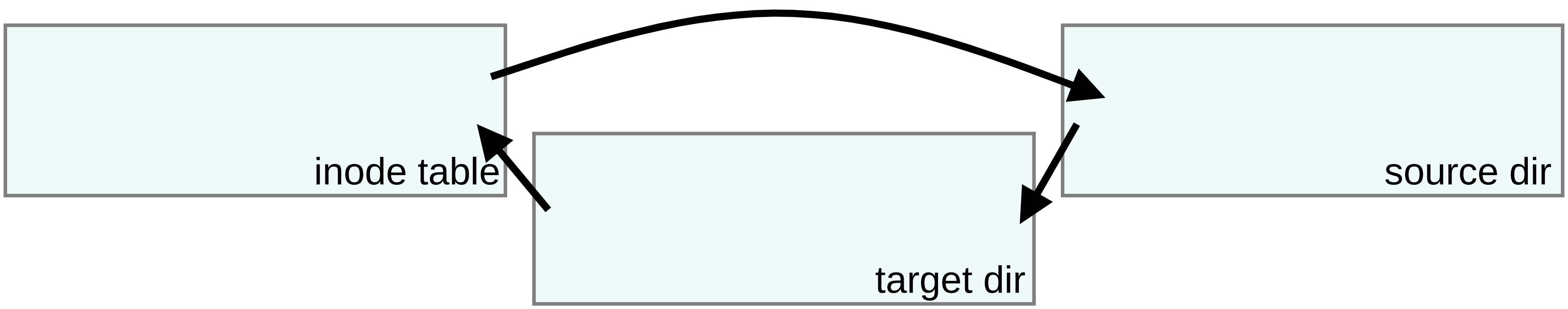
A valid block writeout:



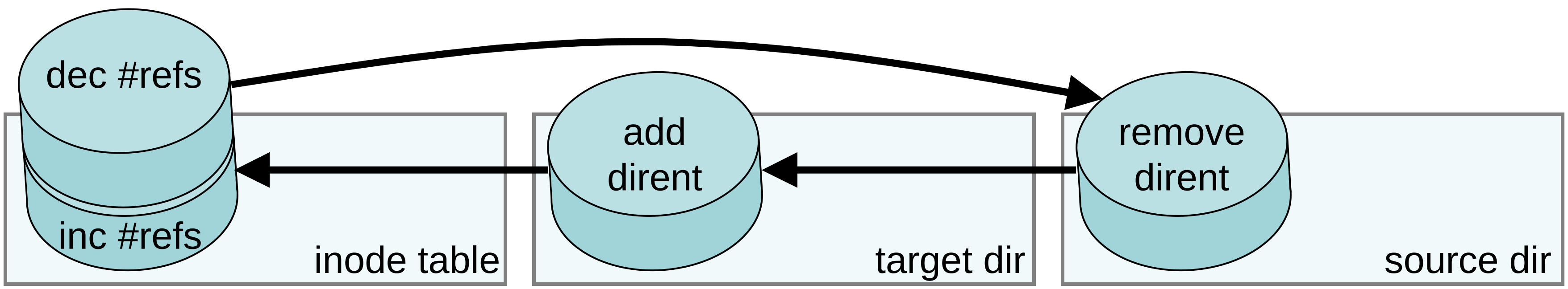
Patch Example: rename() With Soft Updates



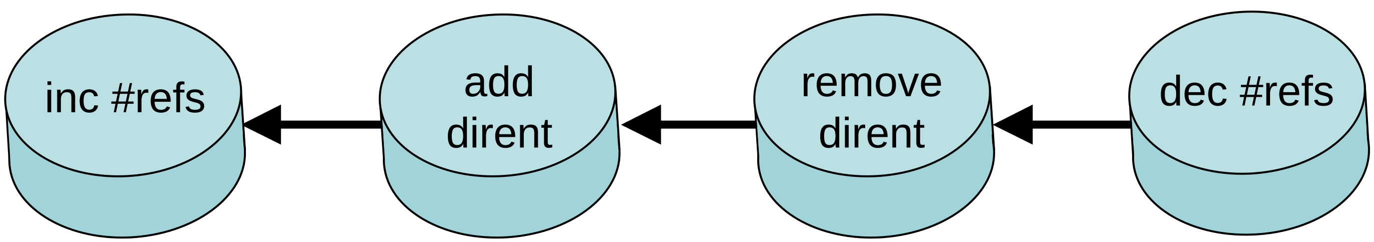
Block level cycle:



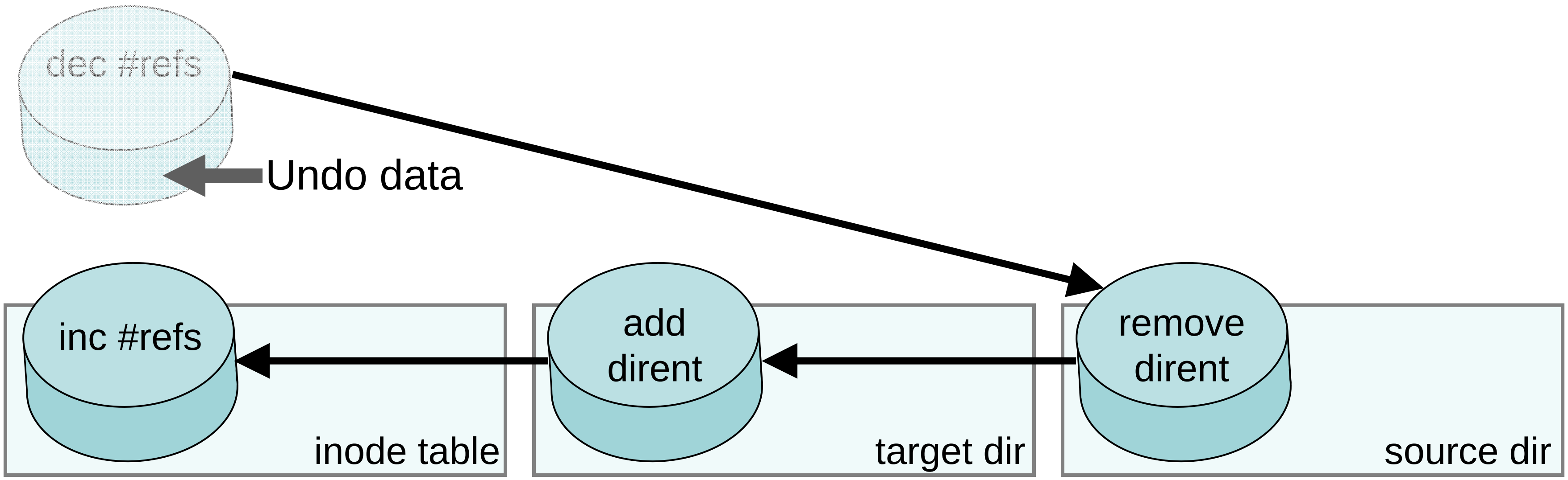
Patch Example: rename() With Soft Updates



Not a *patch level* cycle:

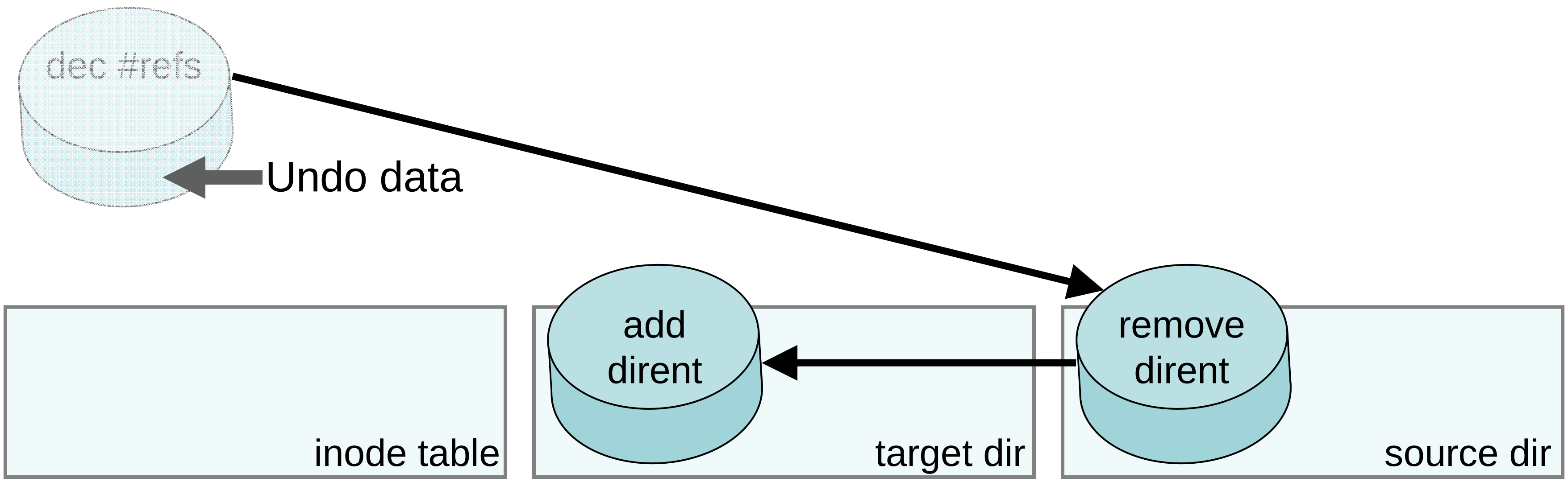


Patch Example: rename() With Soft Updates

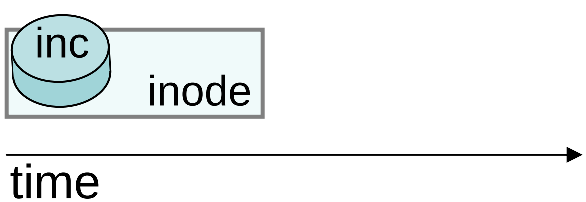


A valid block writeout:

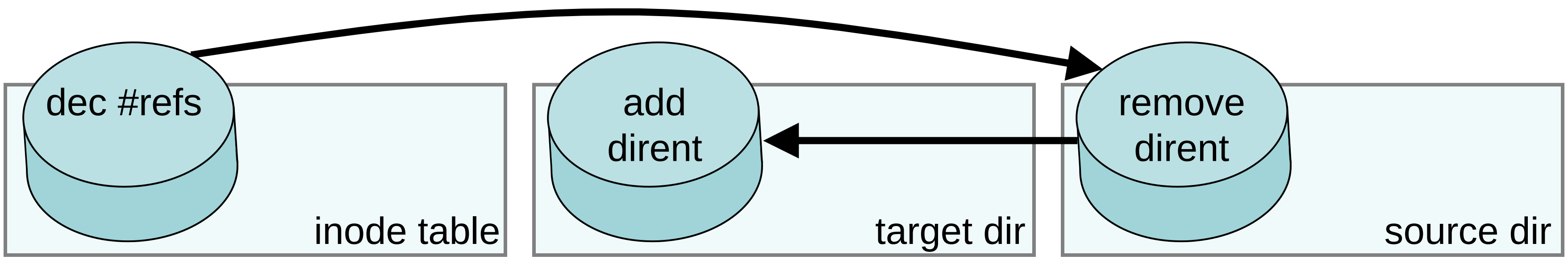
Patch Example: rename() With Soft Updates



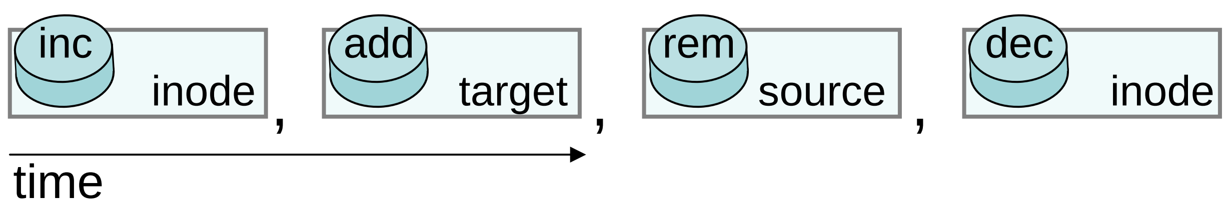
A valid block writeout:



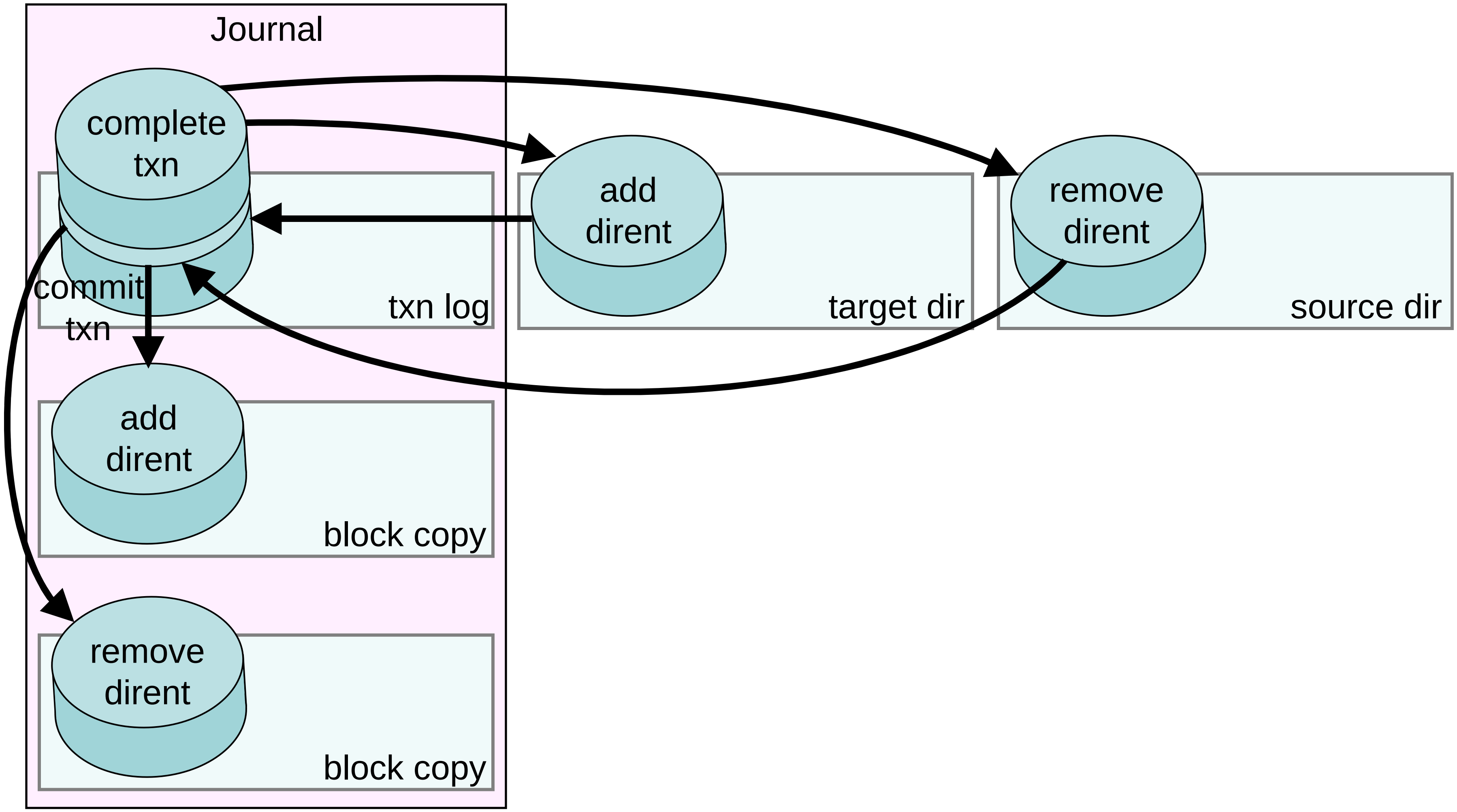
Patch Example: rename() With Soft Updates



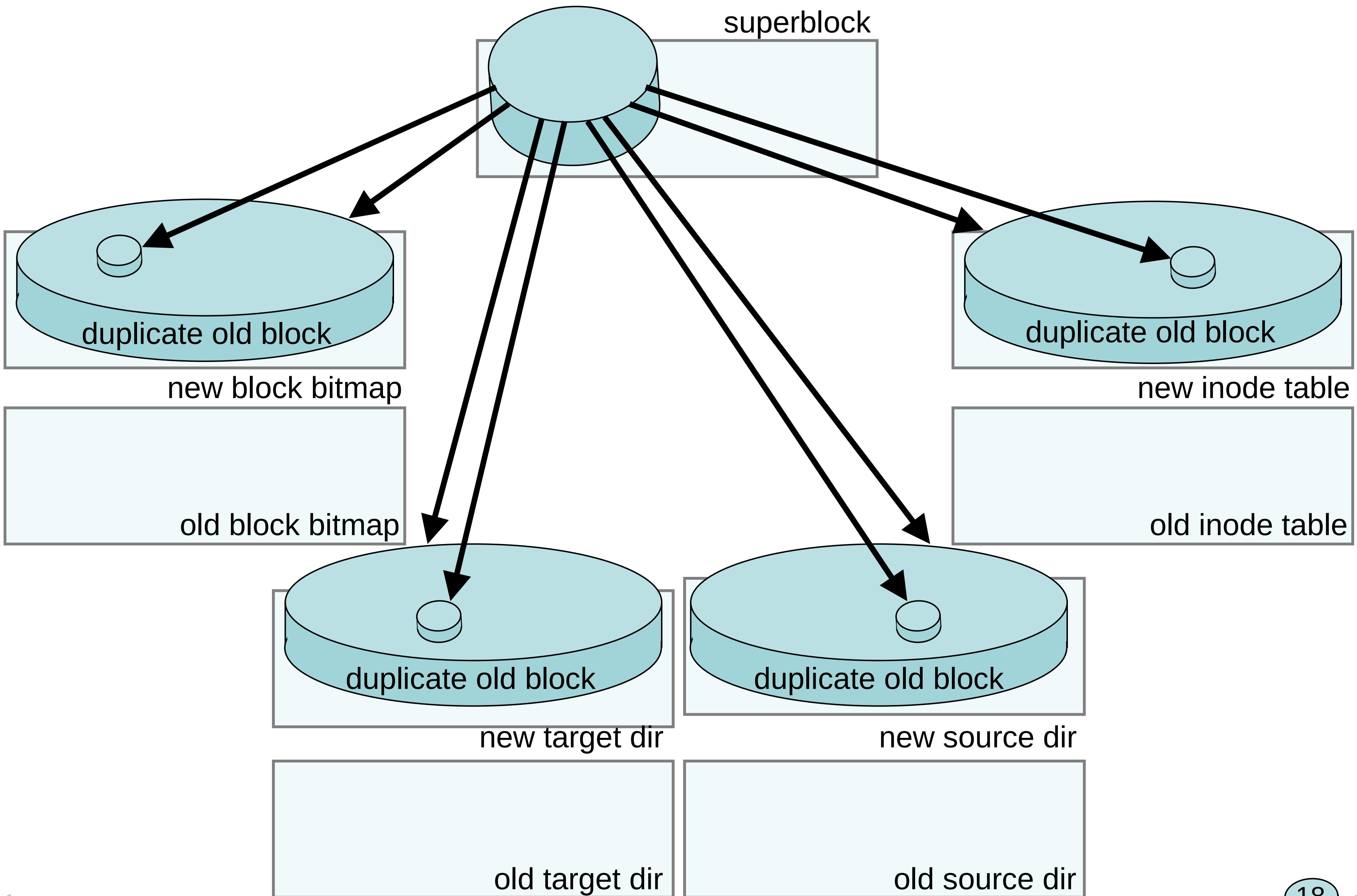
A valid block writeout:



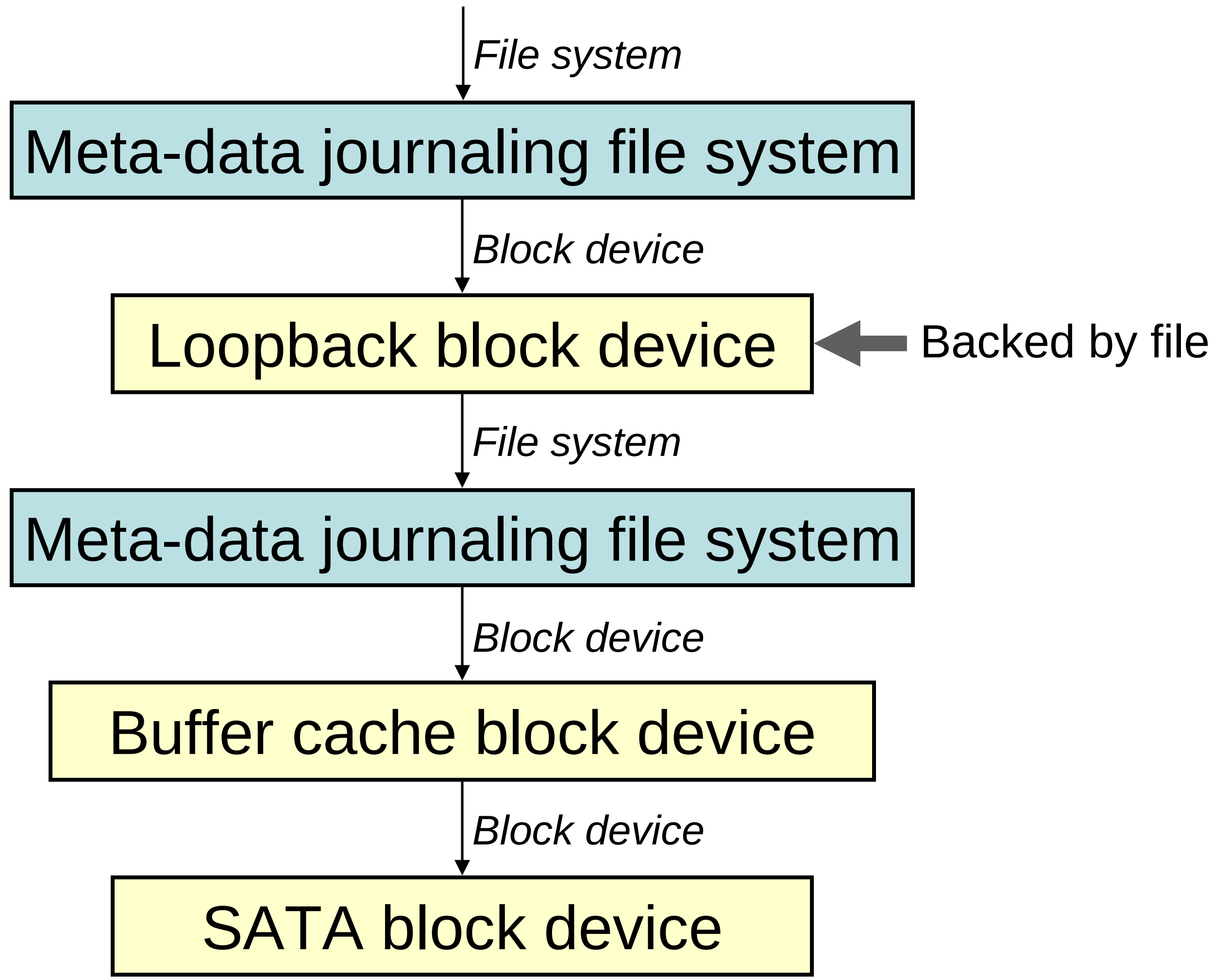
Patch Example: rename() With Journaling



Patch Example: rename() With WAFL



Patch Example: Loopback Block Device



Meta-data journaling file system obeys file data requirements

Patchgroups

Problem

Patches for file systems

Patches for applications

Patch optimizations

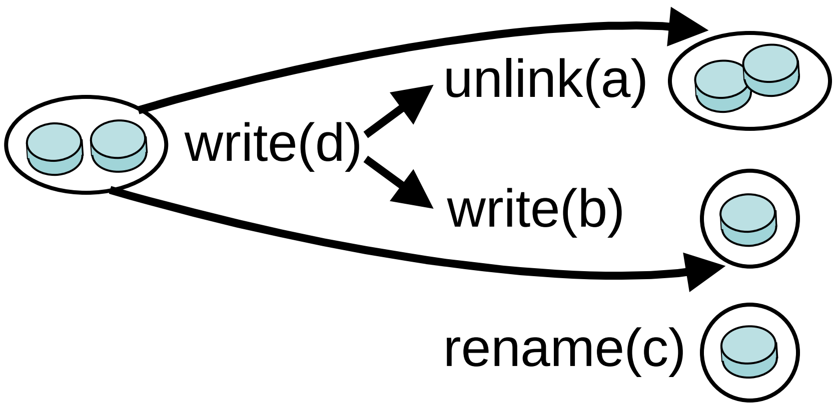
Evaluation

Application Consistency

- Application-defined consistency requirements
  - Databases, Email, Version control
- Common techniques:
  - Tell buffer cache to write to disk immediately (fsync et al)
  - Depend on underlying file system (e.g., ordered journaling)

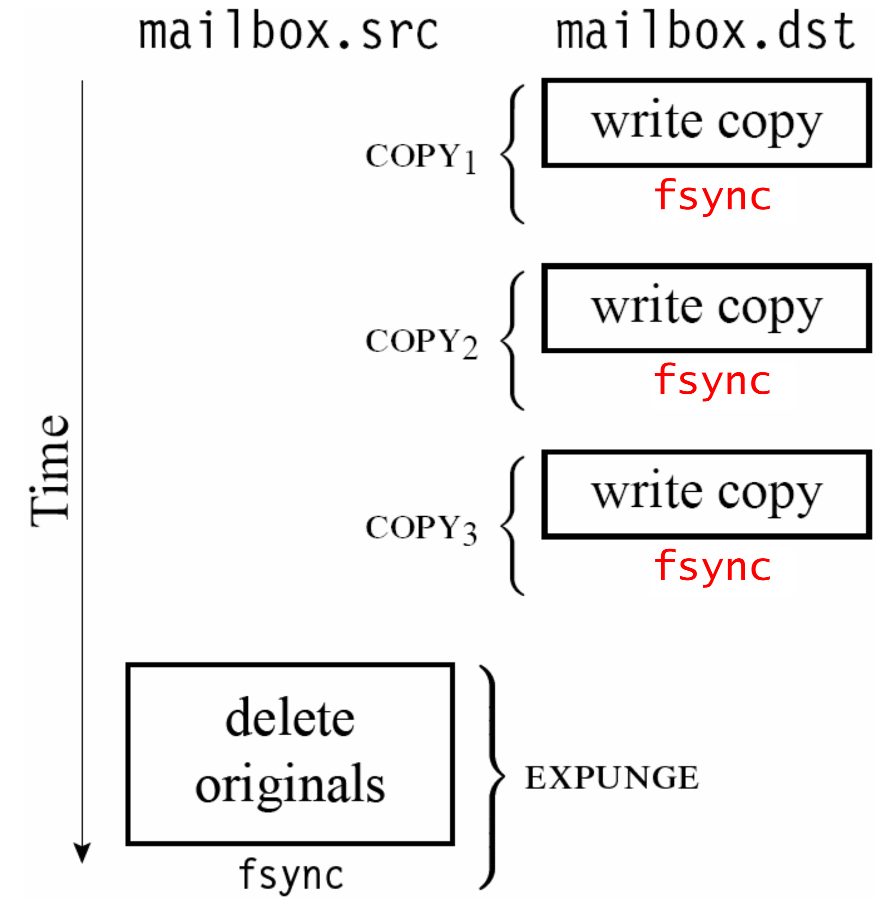
Patchgroups

- Extend patches to applications: *patchgroups*
  - Specify write-before requirements among system calls

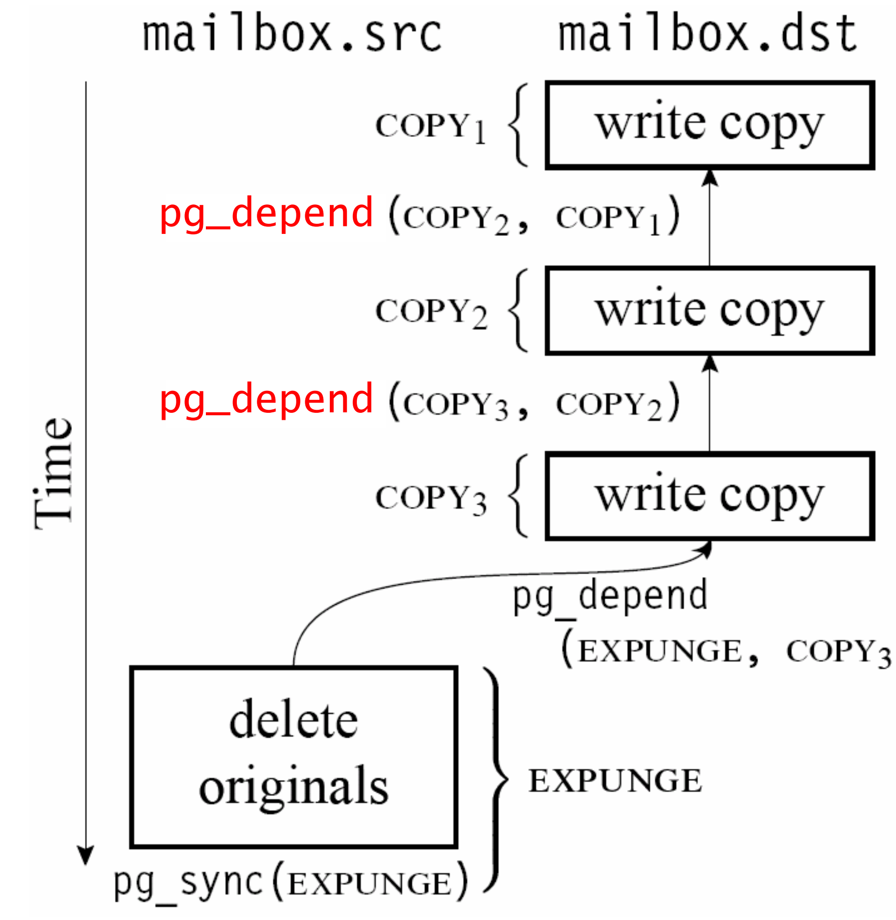


- Adapted gzip, Subversion client, and UW IMAP server

Patchgroups for UW IMAP



Unmodified UW IMAP

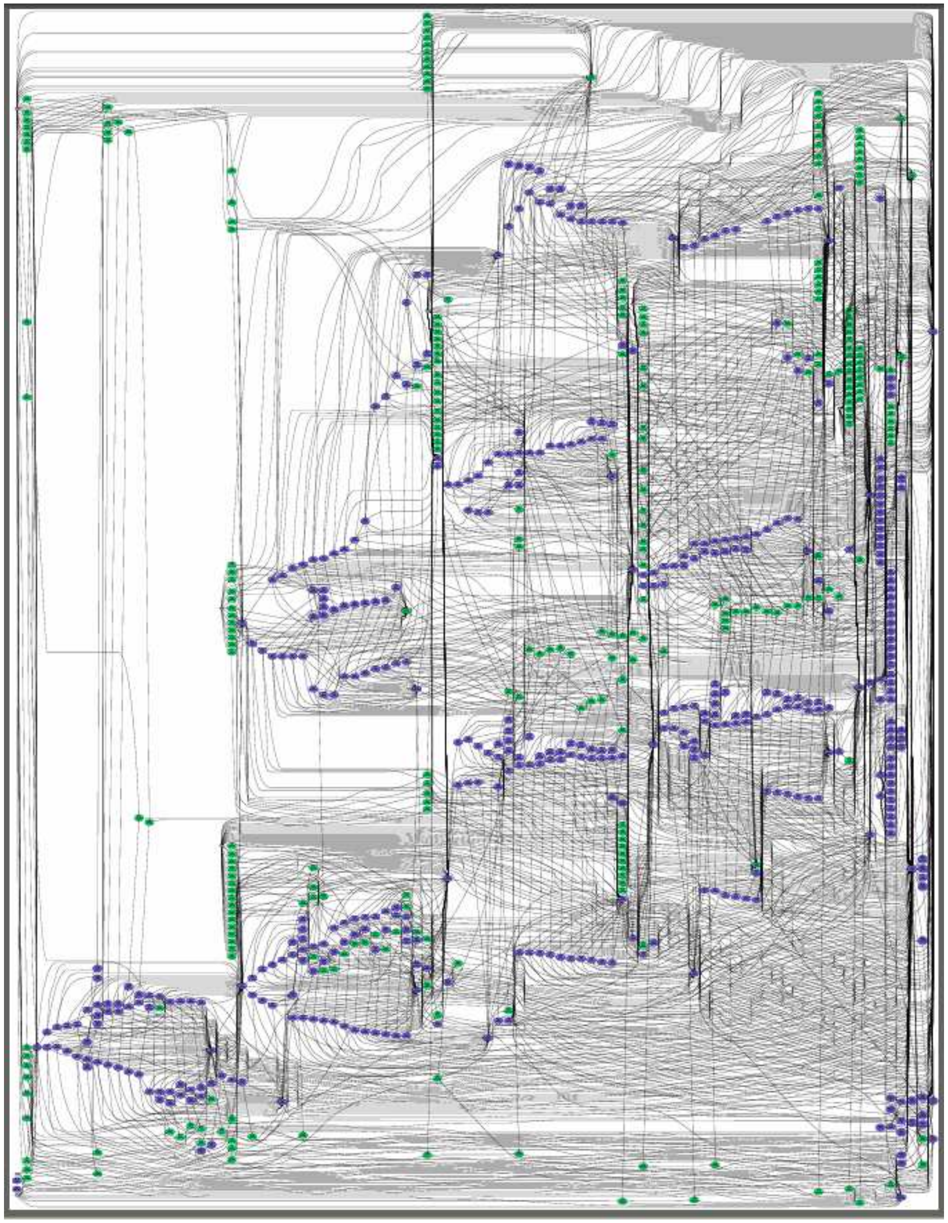


Patchgroup UW IMAP

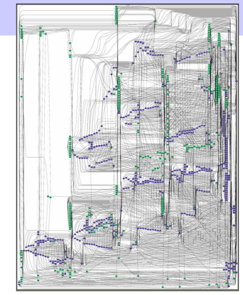
Patch Optimizations

- Problem
- Patches for file systems
- Patches for applications
- **Patch optimizations**
- Evaluation

Patch Optimizations



Patch Optimizations



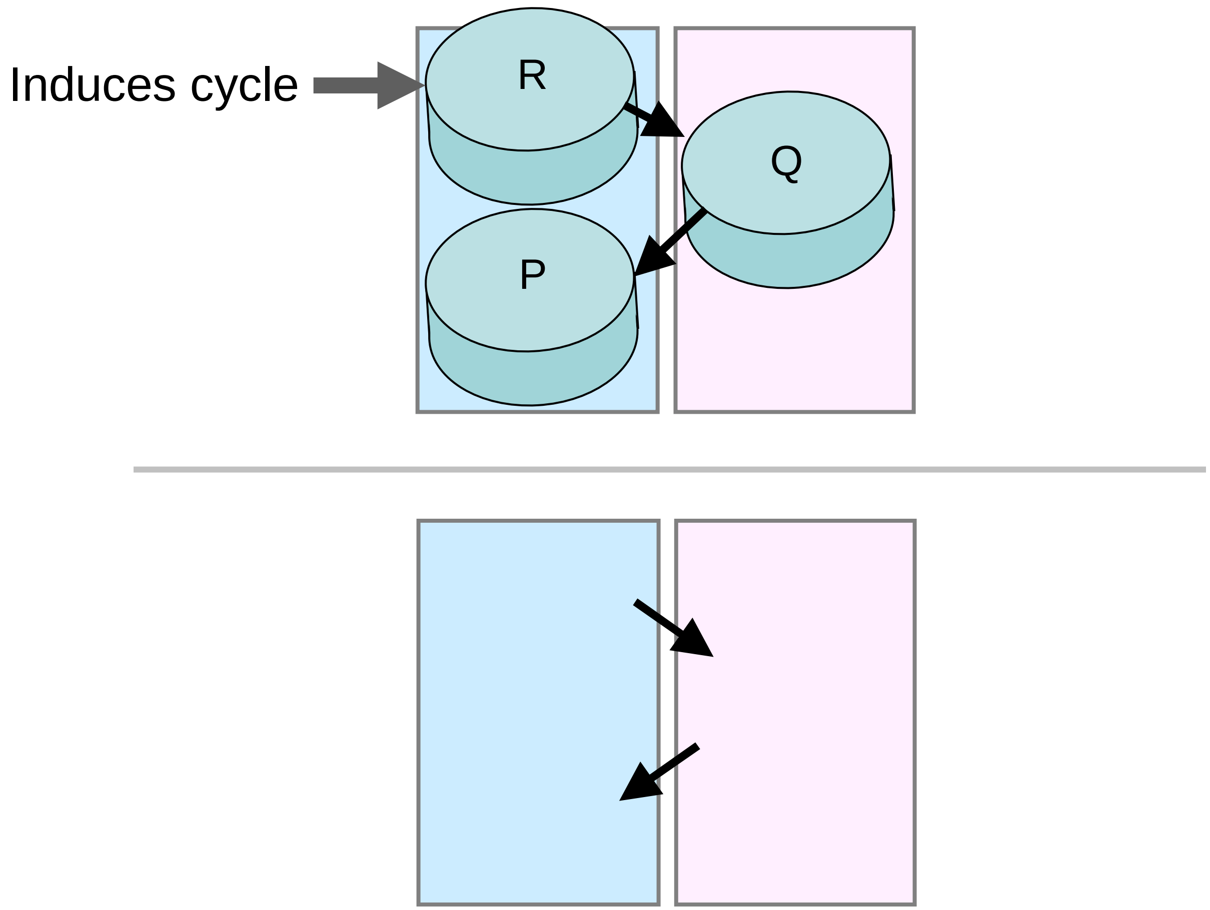
- In our initial implementation:
  - Patch manipulation time was the system bottleneck
  - Patches consumed more memory than the buffer cache
- File system agnostic patch optimizations to reduce:
  - Undo memory usage
  - Number of patches and dependencies
- Optimized Featherstitch is not much slower than Linux ext3

Optimizing Undo Data

- Primary memory overhead: unused (!) undo data
- Optimize away unused undo data allocations?
  - Can’t detect “unused” until it’s too late
- Restrict the patch API to reason about the future?

Optimizing Undo Data

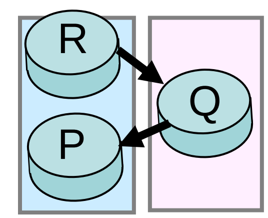
Theorem: A patch that must be reverted to make progress must *induce a block-level cycle*.



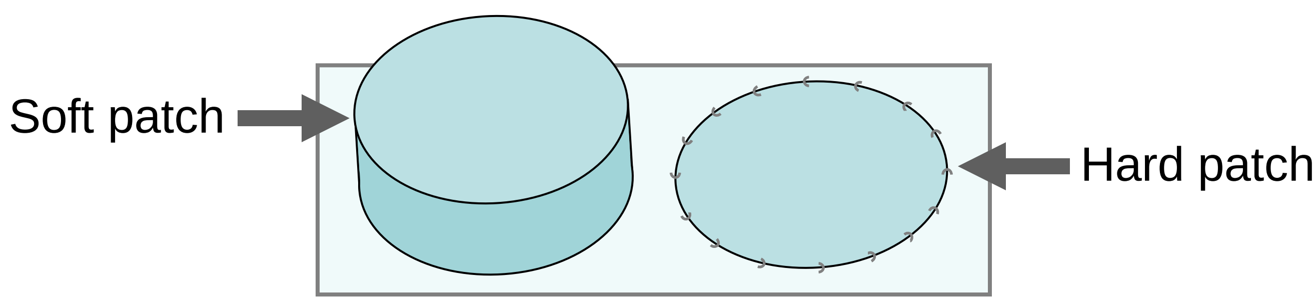
Hard Patches

- Detect block-level cycle inducers when allocating?
  - Restrict the patch API: supply all dependencies at patch creation*

- Now, any patch that will need to be reverted must induce a block-level cycle at creation time

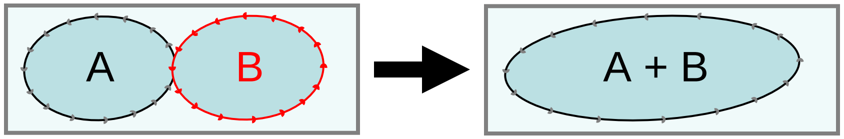


- We call a patch with undo data omitted a **hard patch**. A **soft patch** has its undo data.

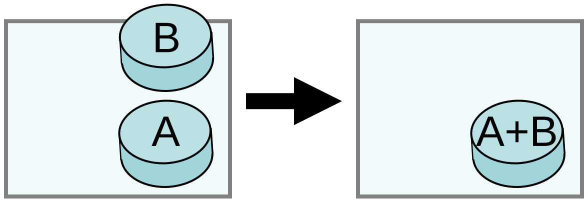


Patch Merging

- Hard patch merging



- Overlap patch merging



Evaluation

Problem

Patches for file systems

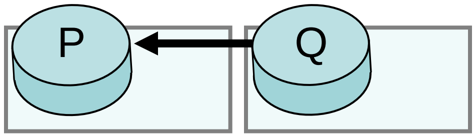
Patches for applications

Patch optimizations

Evaluation

Efficient Disk Write Ordering

- Featherstitch needs to efficiently:
  - Detect when a write becomes durable
  - Ensure disk caches safely reorder writes



- SCSI TCQ or modern SATA NCQ
  + FUA requests or WT drive cache

- Evaluation uses disk cache safely for both Featherstitch and Linux

Evaluation

- Measure patch optimization effectiveness
- Compare performance with Linux ext2/ext3
- Assess consistency correctness
- Compare UW IMAP performance

Evaluation: Patch Optimizations

PostMark

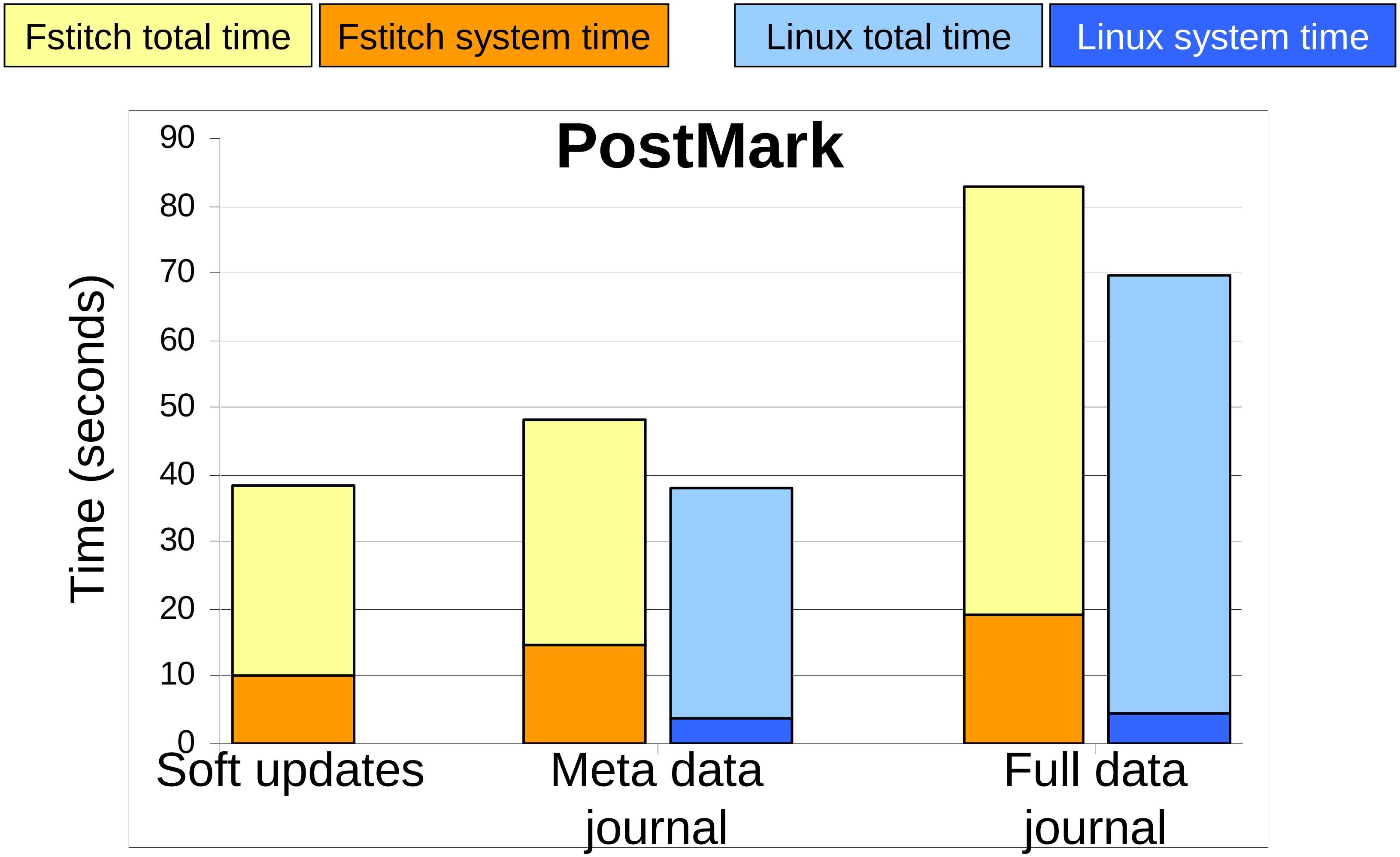
Optimization	# Patches	Undo data	System time
None	4.6 M	3.2 GB	23.6 sec
Hard patches	2.5 M	1.6 GB	18.6 sec
Overlap merging	550 k	1.6 GB	12.9 sec
Both	675 k	0.1 MB	11.0 sec

Evaluation: Patch Optimizations

PostMark

Optimization	# Patches	Undo data	System time
None	4.6 M	3.2 GB	23.6 sec
Hard patches	2.5 M	1.6 GB	18.6 sec
Overlap merging	550 k	1.6 GB	12.9 sec
Both	675 k	0.1 MB	11.0 sec

Evaluation: Linux Comparison



- Faster than ext2/ext3 on other benchmarks
  - Block allocation strategy differences dwarf overhead

Evaluation: Consistency Correctness

- Are consistency implementations correct?
- Crash the operating system at random
- Soft updates:
  - Warning: High inode reference counts (expected)
- Journaling:
  - Consistent (expected)
- Asynchronous:
  - Errors: References to deleted inodes, and others (expected)

Evaluation: Patchgroups

- Patchgroup-enabled vs. unmodified UW IMAP server benchmark: move 1,000 messages
- Reduces runtime by 50% for SU, 97% for journaling

Related Work

- Soft updates [Ganger ’00]
- Consistency research
  - WAFL [Hitz ‘94]
  - ACID transactions [Gal ’05, Liskov ’04, Wright ’06]
- Echo and CAPFS distributed file systems [Mann ’94, Vilayannur ’05]
- Asynchronous write graphs [Burnett ’06]
- xsyncfs [Nightingale ’05]

Conclusions

- Patches provide new write-before abstraction
- Patches simplify the implementation of consistency models like journaling, WAFL, soft updates
- Applications can precisely and explicitly specify consistency requirements using patchgroups
- Thanks to optimizations, patch performance is competitive with ad hoc consistency implementations

Featherstitch source:

http://featherstitch.cs.ucla.edu/

Thanks to the NSF, Microsoft, and Intel.