UNIVERSITY OF CALIFORNIA
Los Angeles

Improving File System Consistency and Durability with Patches and BPFS

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Christopher Cunningham Frost

This version includes some corrections relative to the submitted thesis and is single-spaced

2010

The dissertation of Christopher Cunningham Frost is approved.

Todd D. Millstein

Lixia Zhang

Edmund B. Nightingale

Edward W. Kohler, Committee Chair

University of California, Los Angeles
2010

TABLE OF CONTENTS

List of Figures

ACKNOWLEDGMENTS

I thank Eddie Kohler profoundly, especially for sharing the fun of systems work, sharing his entropy and sense of style, showing me the importance of making software simple and writing concrete, always asking about the why behind the what and how, and his support and inspiration.

Two years and some change ago I received an email that lead to what became BPFS. Thank you, Ed Nightingale and Jeremy Condit, for reaching out, for convincing me it just might work, and, most of all, for the time we've worked together.

Todd Millstein, Lixia Zhang, Adam Meyerson, Richard Korf, Jens Palsberg, Mario Gerla, Majid Sarrafzadeh, and Paul Eggert helped me to grow as a computer scientist in breadth and ability through their support. Liuba Shrira provided support for Featherstitch early on. Valerie Aurora's support, in person, by email, and on the web ([2], among others), instilled excitement about our work in me, the other Featherstitch authors, and the Linux community. Dave Evans shared his curiosity and provided encouragement, support, and guidance throughout my undergrad at UVa.

Shepherds Andrea Arpaci-Dusseau, Geoff Kuenning, and Jeff Mogul; anonymous reviewers from OSDI '06 and '08, SOSP '07 and '09, and USENIX '09; and Bill Bolosky, Mike Dahlin, Jason Flinn, Galen Hunt, Don Porter, and Emmett Witchel provided very useful feedback and guidance on papers that lead to this dissertation. Robert Morris helped make the Featherstitch talk more concrete.

Peter Follett trusted us with the UCLA CSD network and always had a ready ear and interesting network tale. Verra Morgan, Nancy Neymark, Korina Pacyniak, Wenona Colinco, and Steve Arbuckle were always open to help me with what I suspect was an unusually large number of UC and UCLA administrative issues.

I'm grateful for, and thoroughly enjoyed, the many conversations these people shared with me to help me work out the next steps.

Tom Anderson
Justin Cappos
Colin Dixon
Michael Piatek
Arvind Krishnamurthy
– at UW –

Timothy Roscoe
Andrew Bauman
Oriana Riva
Gustavo Alonso
– at ETH Zürich –

Margo Seltzer
Bryan Ford
Remzi Arpaci-Dusseau
Garth Gibson
Jason Flinn
– from as many places as their count –

John Swanson
Vivek Singh
Matthew Gumbel
Marc Eaddy
– at Intel –

Andrew Fikes
Jeff Dean
Wilson Hsieh
Sanjay Ghemawat
Kathy Polizzi
Dana Dickinson
Jelena Pješivac-Grbović
Tony Castellano
John Wilkes
– at Google –

Jiri Schindler
Keith Smith
John Strunk
Chris Small
Jeff Kimmel
Jeff Heller
– at NetApp –

Eddie Kohler
Jennifer Lee
Petros Efsthatopoulos
Andrew de los Reyes
Todd Millstein
Mike Mammarella
Steve VanDeBogart
Brian Chin
Neha Narula
David Chu
– friends from near and far –

Thank you for your friendship.

Roy Shea
David Jurgens
Michael Meisel
Matt Beaumont-Gay
Eric Osterweil
Ani Nahapetian
Gunes Ercal
Ben Greenstein
Martin Lukac
Tom Schoellhammer
Josh Hyman
Akhilesh Singhania
Jamie MacBeth
Ben Titzer
Fernando Pereira
Emanuel Lin
Kirill Minkovich
Neil Tilley
Robert Chen
Adam Harmetz
Tom Lai
Helen Lurie
Joshua Uziel
Mark Fasheh
Manish Singh
Matt Helsley
Mike Ryan
Romy Maxwell
Stephen Sakamoto
Peter Follet
Charlie Fritzius
Richard Guy
Glenn Reinman
Peter Reiher
Todd Millstein
Lixia Zhang
Adam Meyerson
Richard Korf
Jens Palsberg
Mario Gerla
Majid Sarrafzadeh
Paul Eggert
Verra Morgan
Wenona Colinco
Steve Arbuckle
Korina Pacyniak
Nancy Neymark
Anne Dudfield
Marc Rafelski
Damien Ramunno-Johnson
Elaine Wong
Jiin Kim-Daines
Bill Bower
– at UCLA –

Yulis Wardjiman
Jeph Gord
Jesse Moore
Angeliki Kapoglou
Jerome Chang
Denise Chang
Whitney Anderson
Nancy Ram
Dave Malacrida
Tom Velky
Brian's and my Italian neighbor
Andre Brochier
– in LA –

David Mazières

Mike Freedman
Dan Freedman
Andrew Warfield
Brendan Cully

Erik Riedel
Alyssa Henry
Ric Wheeler
Atul Adya
Kirk McKusick
Brad Karp
Florentina Popovici

Dan Peek
Nitin Agrawal
Kiran-Kumar Muniswamy-Reddy
Brandon Salmon

Vijayan Prabhakaran
Geoff Kuenning
Jeff Mogul

Liuba Shrira
Valerie Aurora
Devon Shaw
Rik Farrow
Matthias Felleisen
Gilad Bracha

– who descended upon conferences from all over –

Austin Clements
Dan Ports
Irene Zhang
Max Krohn
Neha Narula

Micah Brodsky
Alex Yip
Russ Cox
Emil Sit
Nikolai Zeldovich
Aleksey Pesterev

Chris Lesniewski-Laas
Jeremy Stribling
Silas Boyd-Wickizer
David Wentzlaff

Robert Morris
M. Frans Kaashoek

Amy Daitch
Emilie Kim
Laura Yip

– from MIT –

Don Porter
Emmett Witchel
Chris Rossbach
– from UT Austin –

Carl Quinn
Tim James
Joel Zacharias
John Callaway
Dmitriy Portnov
Erik Hilsdale
Prasenjit Phukan
Greg Badros
Josh Bloch
Frank Yellin
Rob Konigsberg
Ronald Hughes
Pablo Bellver
Li Moore
Bart Locanthi
Matthieu Devin
Niels Provos
Marius Eriksen
Brian Wickman
Simone McCloskey
Jenny Yuen
– at Google –

Jinyuan Li
Murali Vilayannur
Mayank Rawat
Manju Rajashekhar
Satyam Vaghani
Krishna Yadappanavar
Abhijit Paithankar
Faraz Shaikh
Sandy Sakai
Geoffrey Lefebvre
Tom Phelan
Haripriya Rajagopal
Sudarsan Piduri
Mallik Mahalingam
Irfan Ahmad
Steve Herrod
– at VMware –

Ed Nightingale　　Jeremy Condit
Aaron Schulman　　Matt Renzelmann　　Shriram Rajagopalan
Chris Hawblitzel　　Jay Lorch　　Derrick Coetzee
Orion Hodson　　Galen Hunt　　Barry Bond
Engin İpek　　Ben Lee
Bill Bolosky
Jon Howell　　Jeremy Elson
John Douceur
James Mickens　　Stuart Schechter
Ruzica Piskac
Dave Levin
Maritza Johnson　　Erika Shehan Poole
Rich Draves
Adam Sapek　　Chris Gray
Erika Ross　　Brian Blinn
Brian Chang　　Olga Wichrowska
Ilya Kozavchinsky　　Naomi Hall
TERTL's foam darts
– at Microsoft and in Seattle –

Deric Horn　　Dominic Giampao
– from Apple –

Miklos Szeredi　　Amit Singh　　James Bottomley
Tino Keitel
Asheesh Laroia　　Jack Repenning
– for FUSE, mswatch, and scord –

Jens Axboe　　Wu Fengguang　　Andy Isaacson
Andrew Morton　　Andres Freund　　Cédric Villemain
Andi Kleen　　Rik van Riel　　Ted Ts'o
– from LKML for Featherstitch and libprefetch –

Claudia Kim
Noorhi Kim　　Heyon N. Shin
Mr. William and Mrs. Tina Green
– for their love –

Serge Egelman
Chris Snook
Mike Peck Lisa Dryden
David Chu Wynn Nyane
Shan Wu
Jim Van Dyke
Jessi Greer Jon McCune
Clio Kakoulli
Meg Olson
Angela Widhalm Paul Bui Greg Joiner
Arthur Orton Beth Dykes Dave Brown
Dave Evans
– from undergrad –

Michael Nutt
Mr. Jerry and Mrs. Dorrie Nutt
Emily McCann
Kelsey Clark Matt Terry
Christy Hales Luda Shtessel Lori Graves Nicole Loo
Sara Beth DeLisle
Sarah Sandy Steigner
Mark DeLong Morgan Qualls Dag Rowe
Mirandy Hughes Danae Tinsley Boyd Kathryn Boyd Brekle
Kevin Burge Tejus Ujjani
Brad Powell Alan Duggan
Rob Ingram
Mark Spencer
Susanna Phillips
Dr. Macon and Mrs. Barbara Phillips
Rod Montgomery Chris Adams
Rosalind Marie
Pat and Roger Schwerman
– from even earlier –

Thank you to those who were entertainingly odd, too: from the man who visited me at UCLA, convinced that he could prove P = NP based on a paper I'd written years before, to the Swiss guy who sent me a curious message about my snow attire, to the strange phone call from the CTO who wasn't really the CTO (but who wanted to hire me).

Thank you to the members of our lab, TERTL, who have become wonderful friends, collaborators, instigators, and accomplices.

. . ^Mike Mammarella, Christina Oran, Andrew de los Reyes, Allison de los Reyes, Lei Zhang, Yonnie Louie, Shant Hovsepian, Andrew Matsuoka, Steve VanDeBogart, Melissa Barshop, Petros Efsthatopoulos, Brian Chin, Dan Marino, Alex Warth, Carolyn Morse, Shane Markstrum, Yinqing Mao, Jacob Lacouture, Nikitas Liogkas, Ru-Gang Xu, Mike Emmi, Jeff Fischer, Nithya Ramanathan, Rob Nelson, Milan Stanojević, Manav Mital, Tom Bergan, Dan Hipschman, Dero Gharibian, Michael Gray, . .

. . Hesam Samimi, Kousha Najafi, and Rafit Izhak-Ratzin. Pierre Ganty, Roman Manevich, Elaine Render, . . .

. . . and Jeff Vaughan. Eddie Kohler, Todd Millstein, and Rupak Majumdar. Vi and Emacs. And Horrik Sicles and Fishberos.$

Laura Poplawski Ma, Michael Shindler, Mark Painter, Aprotim Sanyal, Lauren Sanyal, Andrew Howe, Taylor Haight, Evan Stade, and Jenny Wang, thank you, too.

Jennifer Lee, thank you for your love, encouragement and support, insight, and understanding.

Finally, Mom, Dad, Hampton, and the Frost family, Erin, Hunter, Meghan, Marian, Max, Iris, Mary, David, Anne, Mark, Laurie, Charlie, Beth, and Papa, thank you for your love and support. Mom and Dad, thank you, too, for your guidance and your belief in me.

Chapter 2 is a revision of “Generalized File System Dependencies,” published in the Proceedings of the 2007 ACM Symposium on Operating Systems Principles (SOSP), ©ACM, 2007. It is joint work done with Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang, and was supported by the National Science Foundation, Microsoft Research, and Intel.

Chapter 3 is an extension of “Better I/O Through Byte-Addressable, Persistent Memory,” published in the Proceedings of the 2009 ACM Symposium on Operating Systems Principles (SOSP), ©ACM, 2009. It is joint work done with Jeremy Condit, Edmund B. Nightingale, Engin Ipek, Doug Burger, Benjamin Lee, Derrick Coetzee, and Eddie Kohler, while I was at Microsoft Research and at UCLA, and was supported by Microsoft Research, the National Science Foundation, the Alfred P. Sloan Foundation, and Intel.

VITA

1981	Born, Huntsville, Alabama
2004	Bachelor of Science, Computer Science Bachelor of Arts, Mathematics University of Virginia
2004–2005	Departmental Fellowship Computer Science Department University of California, Los Angeles
2005–2006	Teaching Assistant Computer Science Department University of California, Los Angeles
2006	Master of Science, Computer Science University of California, Los Angeles
2005–2010	Graduate Research Assistant Eddie Kohler, Computer Science Department University of California, Los Angeles

PUBLICATIONS

Condit, J., Nightingale, E., Frost, C., Ipek, E., Burger, D., Lee, B., and Coetzee, D. 2010. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the Twenty-Second ACM Symposium on Operating Systems Principles* (Big Sky, Montana, October 11–14, 2009). SOSP '09. ACM, New York, NY, 147–160.

de los Reyes, A., Frost, C., Kohler, E., Mammarella, M., and Zhang, L. 2005. The KudOS Architecture for File Systems. Work in progress session, Twentieth ACM Symposium on Operating Systems Principles. (Brighton, United Kingdom, October 23–26, 2005). SOSP '05. ACM, New York, NY.

Frost, C. and Millstein, T. 2005. Featherweight JPred. UCLA CS Tech Report CSD-TR-050038. (October 2005).

Frost, C. and Millstein, T. 2006. Modularly Typesafe Interface Dispatch in JPred. The 2006 International Workshop on Foundations and Developments of Object-Oriented Languages (Charleston, South Carolina, January 14, 2006). FOOL/WOOD '06. ACM, New York, NY.

Frost, C., Mammarella, M., Kohler, E., de los Reyes, A., Hovsepian, S., Matsuoka, A., and Zhang, L. 2007. Generalized File System Dependencies. In *Proceedings of the Twenty-First ACM Symposium on Operating Systems Principles* (Stevenson, Washington, October 14–17, 2007). SOSP '07. ACM, New York, NY, 307–320.

Millstein, T., Frost, C., Ryder, J., and Warth, A. 2009. Expressive and Modular Predicate Dispatch for Java. In *Proceedings of the ACM Transactions on Programming Languages and Systems* 31(2):7:1–54. (February 2009).

VanDeBogart, S., Frost, C., and Kohler, E. 2009. Reducing Seek Overhead with Application-Directed Prefetching. In *Proceedings of the 2009 USENIX Annual Technical Conference* (San Diego, California, June 14–19 2009). USENIX '09. USENIX Association, Berkeley, CA, 299–312.

ABSTRACT OF THE DISSERTATION

Improving File System Consistency and Durability with Patches and BPFS

by

Christopher Cunningham Frost

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2010

Professor Edward W. Kohler, Chair

This dissertation improves the consistency and durability guarantees that file systems can efficiently provide, both by allowing each application to choose appropriate trade-offs between consistency and performance and by dramatically lowering the overheads of durability and consistency using new hardware and careful file system design.

We first describe a new abstraction, the *patch*, which represents a write to persistent storage and its ordering requirements. This abstraction allows file system modules to specify ordering guarantees without simultaneously requesting more expensive, immediate durability. Algorithmic and data structure optimizations make this abstraction practical, and our patch-based file system implementation is performance-competitive with similarly-reliable Linux ext2 and ext3 configurations. To illustrate the benefits of *patchgroups*, an application-accessible version of patches, we apply them to improve the performance of the UW IMAP server by over 13 times and to make file handling in gzip robust to crashes.

In the second part of this work we investigate using upcoming byte-addressable, persistent memory technologies – in particular, phase change memory – in place of disks and flash to reduce the costs of enforcing ordering constraints and providing durability and atomicity. We describe a new file system, *BPFS*, that commits *each file system operation* synchronously and atomically. BPFS exploits byte-addressability, improved throughput and latency, and our new atomic write primitive to eliminate copy-on-writes that have until now been required to implement shadow paging. Our evaluation shows that, because of our optimizations, BPFS provides its exceptionally stronger guarantees on phase change memory without lowering the throughput that today's file systems achieve on disks.

Chapter 1

Introduction

This dissertation develops techniques that increase the resiliency of file systems in the face of hardware and software crashes.

File systems are the portions of an operating system primarily responsible for storing data *durably*: that is, so that it will outlast the current boot. They provide to applications the interface of a collection of files, often arranged in a hierarchy. To implement this interface, file systems store a collection of data structures in a persistent medium; typically, this is a hard disk drive or flash memory.

However, a combination of factors make it difficult to modify these persistent data structures without losing or corrupting already-durable data.

First, a software or hardware fault may halt execution at any time, including during a sequence of writes to persistent storage. Therefore the file system must ensure that its persistent storage is internally consistent at all times. Otherwise, a halt may cause the file system to lose or corrupt durable data.

Second, the interfaces provided by persistent storage hardware support only sequences of small, non-atomic writes. However, many file system operations can modify multiple persistent data structures, and should do so atomically. Thus, the file system must ensure consistency even at fine granularities.

And third, persistent storage hardware often operates at latencies far higher and throughputs far lower than CPUs, RAM, and system buses. This performance relationship makes efficiency a principal concern in how file systems interact with persistent storage.

For these reasons, a primary focus of file system research has been the design of data structures that provide consistency and durability without losing too much efficiency. Early file system designs achieved durability, and a degree of consistency, by updating data structures on disk synchronously with those in RAM. Due to technological factors – the performance improvements of mechanical disks vs. solid state CPUs and RAM – this approach has become prohibitively expensive. Disks are now many orders of magnitude slower than CPUs. Most recent file systems therefore provide consistency with only weak durability guarantees. Journaling file systems, for example, group the disk changes made by a file system operation into larger atomic units that span many operations, but delay commits, meaning the disk changes for an operation may not be committed until some time after the operation returns to the calling process. Additionally, the consistency guarantees that file systems provide to applications are fixed at the time of the file system's design. For example, the default file systems for most operating systems provide strong ordering and atomicity guarantees about the data structures internal to the file systems and weak to no guarantees for application data. Applications are unable to change these guarantees other than through (expensive) requests for immediate durability. This leaves some applications without efficient options for the consistency guarantees they need and some with costly guarantees they do not require. To summarize, current file systems are limited to providing inflexible consistency and weak durability guarantees.

These limitations force the designers of applications to make trade-offs among several dimensions, including consistency, durability, latency, throughput, portability, complexity, and application size. So much engineering effort is required to interact with the file system both robustly and efficiently that many applications give up robustness; but even robust applications are often only robust on a minority of their supported platforms. For example, the client for the Subversion version control system implements application-level journaling to protect itself against application and system crashes, but the journal implementation itself is only robust against system crashes on (some) Linux file systems – the developers decided that robustness on all platforms would make the client too slow (§2.3.2).[1] Techniques that achieve robustness on all platforms, however, often severely limit overall application performance. For example, the initial use of the SQLite database in the Firefox web browser forced file-system-wide immediate durability as part of each page load on the standard Linux file systems [6, 38].

This dissertation addresses these file system limitations in two ways. First, it shows how a new abstraction enables file systems and applications to efficiently add consistency guarantees at run time. Second, it shows how a combination of new storage technologies and file system techniques can provide fast synchronous and atomic file system operations.

Chapter 2 describes a new abstraction, the *patch*, that allows file system modules to explicitly express their precise requirements about the order in which writes are made durable. We have developed a storage system based on patches, Featherstitch, and have found that patches allow file system software to work more flexibly. For example, by exposing patches to applications, through the *patchgroup* interface, applications can obtain their precise write ordering requirements without resorting to requests for immediate durability. At the same time, we have also found that the run-time costs to maintain these abstractions can easily overwhelm the system. This dissertation describes the data structure and algorithmic optimizations that make the performance of these new abstractions practical. In our experiments, these optimizations reduce the amount of memory allocated for the largest source of patch overhead by at least 99% and reduce execution time by up to 80%. We find that the performance of Featherstitch file systems with these optimizations is on par with Linux file systems that provide similar consistency guarantees, while Featherstitch also provides the additional flexibility of patches.

[1]Consistency can be difficult to provide, however. Correctness requires that each command be idempotent, but many times the implementations have turned out not to be [68, 69].

Although the abstractions described in Chapter 2 allow applications to add consistency guarantees without having to micromanage writes to persistent storage, the inherent costs of imposing consistency guarantees for disks and flash remain. Further, applications must still make due with only infrequent durability guarantees: file system operations typically commit asynchronously and commit intervals are rarely shorter than billions of CPU instructions. Chapter 3 investigates the use of upcoming persistent memory technologies in place of disks and flash storage. One could run existing file systems on these technologies, but file systems would still be limited to providing current weak durability guarantees. More frequent commits would cause them to write many times more data. We decided, instead, to try to use the extra capabilities and performance to provide far stronger durability guarantees than current file systems provide. Each file system operation in the resulting BPFS file system commits synchronously and atomically. (Note that unless specified otherwise, references to atomicity in this dissertation refer specifically to failure atomicity.) Conventional file systems would be too slow to provide these guarantees, but this dissertation shows that careful design and several optimizations can make them practical. The addition of these guarantees significantly simplifies the persistent storage interface that file systems provide to applications.

The contributions of this dissertation include:

– The patch optimizations that turn the patch and patchgroup abstractions into a practical file system implementation technique;

– The evaluation of patchgroups;

– The design of new file system techniques for byte-addressable, persistent memory technologies;

– The development of the BPFS file system, which implements these techniques; and

– The evaluation of BPFS and the individual file system techniques.

This dissertation describes the SOSP 2007 version of Featherstitch and the FUSE implementation of BPFS. These releases are available online for download at `http://featherstitch.cs.ucla.edu/` and `http://bpfs.cs.ucla.edu/`.

Chapter 2

Featherstitch

Write-before relationships, which require that some changes be committed to stable storage before others, underlie every mechanism for ensuring file system consistency and reliability from synchronous writes to journaling. (Journaling logs a set of intended changes before committing the changes, so that the changes become durable atomically even if the system crashes.) Featherstitch is a complete storage system[1] built on a concrete form of these relationships: a simple, uniform, and file system agnostic data type called the *patch*.

Featherstitch's API design and performance optimizations make patches a promising implementation strategy as well as a useful abstraction.

A patch represents both a change to disk data and any *dependencies* that change has on other changes. Patches were initially inspired by BSD's soft updates dependencies [17], but whereas soft updates implements a particular type of consistency and involves many structures specific to the UFS file system [42], patches are fully general, specifying only how a range of bytes should be changed. This lets file system implementations specify a write-before relationship between changes without dictating a write order that honors that relationship. It lets storage system components examine and modify dependency structures independent of the file system's layout, possibly even changing one type of consistency into another (e.g., soft updates into journaling). It also lets applications modify patch dependency structures, thus defining consistency policies for the underlying storage system to follow.

Since these modules explicitly specify their write ordering requirements with patches, multiple modules can cooperate to specify overall dependency requirements by passing patches back and forth. Patches can represent any of the consistency mechanisms currently proposed [16, 35, 82, 87]; even combinations of consistency mechanisms can comfortably coexist.

Applications likewise have few mechanisms for controlling buffer cache behavior in today's systems, and robust applications, including databases, mail servers, and source code

[1] In this chapter, we use the term "storage system" to refer to the software layer that implements file system services in an operating system. This helps to distinguish it from the term "file system," which we use to refer to individual on-disk data layouts and the specific modules within storage systems that implement them.

management tools, must choose between several mediocre options. They can accept the performance penalty of expensive system calls like `fsync` and `sync`, which request that the storage system fall back to slow synchronous writes, or use tedious and fragile sequences of operations that assume particular file system consistency semantics like the data-metadata ordering guarantees provided by some modes of the Linux ext3 and ext4 file systems. *Patchgroups*, our example user-level patch interface, export to applications some of patches' benefits for kernel file system implementations and extensions. Modifying an IMAP mail server to use patchgroups instead of `fsync` required only localized changes. The result both meets IMAP's consistency requirements on any reasonable patch-based file system and avoids the performance hit of full synchronization. Instead of the mail server forcing immediate durability as part of each IMAP command to implement the IMAP consistency guarantees, with patchgroups the buffer cache is able to, for example, reduce the number of disk seeks by coalescing data, inode, and free block bitmap disk writes across many IMAP commands.

Production file systems use system-specific optimizations to achieve consistency without sacrificing performance; we had to improve performance in a general way. A naïve patch-based storage system scaled terribly, spending far more space and time on dependency manipulation than conventional systems. However, optimizations reduced patch memory and CPU overheads significantly. A PostMark test that writes approximately 3.2 GiB of data allocates 75 MiB of memory throughout the test to store patches and soft-updates-like dependencies, less than 3% of the memory used for file system data and about 1% of that required by unoptimized Featherstitch. Room for improvement remains, particularly in system time, but Featherstitch outperforms equivalent Linux configurations on many of our benchmarks; it is at most 30% slower on others.

The contributions of the Featherstitch work are the patch model and module system designs, the design of the patch interface and the implementation techniques that make patches an efficient abstraction, the patchgroup mechanism that exports patches to applications, and several individual Featherstitch modules, such as the journal and buffer cache.

There were several notable challenges in developing Featherstitch. It took several iterations to refine the patch API to support rearranging existing patches while remaining efficient. The more invariants we could guarantee about patches, the more strongly we could reason about the future of a patch. While this can open up more optimization opportunities, enforcing too many invariants would restrict the API and make it not flexible enough for some modules. Likewise, we had to strike the right balance of power and safety in the userspace-visible patchgroup interface. Here, we had to be more restrictive, since we must be robust despite the behavior of arbitrary user processes, but we still wanted to provide the maximum level of functionality while protecting the kernel. Another challenge was working with Linux's disk subsystem and on-disk caches to preserve the required block write orderings all the way to the physical disk media. Both Linux's disk scheduler (even, it turns out, the "no-op" scheduler) and on-disk write-back caches may reorder writes; changing that property can be anywhere from impossible to merely performance-degrading. Instead, we designed Featherstitch to work with these systems as they are, yet still without substantial performance penalties.

Featherstitch is joint work with Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. I am responsible for the patch data structure and algorithm optimizations and for the patchgroup application case studies. Mike Mammarella designed most of the module system [39] and helped introduce the patch concept. Our paper on Featherstitch was published in SOSP 2007 [14], where it received an audience choice award.

In this chapter, Section 2.1 first describes patches abstractly, then states their behavior and safety properties, and gives examples of their use. Section 2.2 then motivates, describes, and reasons about the correctness of our optimizations that make patches a practical abstraction. Section 2.3 introduces patchgroups and discusses three case studies. Section 2.4 describes how the Featherstitch architecture is decomposed into modules, and presents several of the more interesting and useful modules. Section 2.5 describes the integration of Featherstitch with the Linux kernel. Then Section 2.6 evaluates our work by measuring the effectiveness of our optimizations and comparing the performance of Featherstitch and Linux-native file system implementations. Finally, Section 3.7 describes related work and Section 3.8 concludes.

2.1 Patch Model

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

2.1.1 Disk Behavior

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

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

A **patch** models any change to block data. Each patch applies to exactly one block, so a change that affects *n* blocks requires at least *n* patches to represent. Each patch is either **committed**, meaning written to disk; **uncommitted**, meaning not written to disk; or **in flight**, meaning in the process of being written to disk. The intermediate in-flight state models reordering and delay in lower storage layers; for example, modern disks often cache writes to add opportunities for disk scheduling. Patches are created as uncommitted. The operating system moves uncommitted patches to the in-flight state by writing their blocks to the disk controller. Some time later, the disk writes these blocks to stable storage and reports success. When the operating system receives this acknowledgment, it commits the relevant patches. Committed patches stay committed permanently, although their effects can be

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

Figure 2.1: Patch notation.

undone by subsequent patches. The sets $\boldsymbol{C}$, $\boldsymbol{U}$, and $\boldsymbol{F}$ represent all committed, uncommitted, and in-flight patches, respectively.

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

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

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

Disks perform better when allowed to reorder requests, so storage systems try to keep many blocks in flight. A block write will generally put all of that block's uncommitted patches in flight, but a storage system may, instead, write a *subset* of those patches, leaving some of them in the uncommitted state. As we will see, this is sometimes required to preserve write-before relationships.

We intentionally do not specify whether the underlying persistent storage device (e.g., the disk) writes blocks atomically. Some file system designs, such as soft updates, rely on block write atomicity, where if the disk fails while a block B is in flight, B contains either the old data or the new data on recovery. Many journal designs do not require this, and include recovery procedures that handle in-flight block corruption – for instance, if the memory holding the new value of the block loses coherence before the disk stops writing [70]. Since patches model the write-before relationships underlying these journal designs, patches do not provide block atomicity themselves, and a patch-based file system with soft-updates-like dependencies should be used in conjunction with a storage device that provides block atomicity.

2.1.2 Dependencies

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

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

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

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

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

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

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

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

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

2.1.3 Dependency Implementation

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

Featherstitch maintains each block in its dirty state, including the effects of all uncommitted patches. However, each patch carries **undo data**, the previous version of the block data altered by the patch. If the buffer cache writes block B, but cannot include patch p in the write, it *reverts* the patch, which swaps the new data on the buffered block and the previous version in the undo data. Once the block is written, the system will re-apply the patch and, when allowed, write the block again, this time including the patch. Some undo mechanism is required to break potential block-level dependency cycles, as shown in the next section. We considered alternate designs, such as maintaining a single "old" version of the block, but per-patch undo data gives file systems the maximum flexibility to create patch structures. However, many of our optimizations avoid storing unnecessary undo data, greatly reducing memory usage and CPU utilization.

Figure 2.1 summarizes our patch notation.

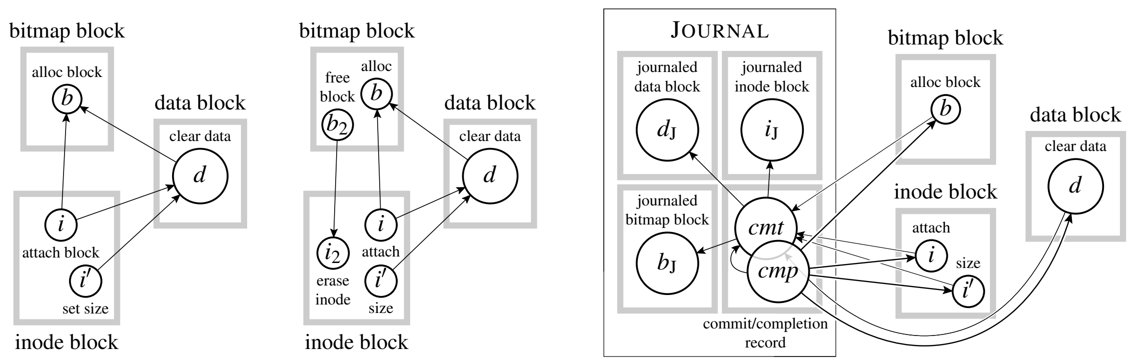
2.1.4 Examples

This section illustrates patch implementations of two widely-used file system consistency mechanisms, soft updates and journaling. Our basic example extends an existing file by a single block – perhaps an application calls `ftruncate` to append 512 zero bytes to an empty file. The file system in this example is based on Linux's ext2, an FFS-like[2] file system with inodes and a free block bitmap. In such a file system, extending a file by one block requires (1) allocating a block by marking the corresponding bit as "allocated" in the free block bitmap, (2) attaching the block to the file's inode, (3) setting the inode's size, and (4) clearing the allocated data block. These operations affect three blocks – a free block bitmap block, an inode block, and a data block – and correspond to four patches: b (allocate), i (attach), i' (size), and d (clear).

With soft updates, the final patch arrangement will consist of just these four patches, with some dependencies between them designed to preserve important invariants on the disk at all times. The correct dependencies are easy to determine, however, by following some simple rules also used in the original soft updates implementation. After going through the simple example, we also examine a variation that produces a block-level cycle, and contrast soft updates patches with the BSD implementation.

With journaling, on the other hand, we will end up with a much more complex-looking patch arrangement: since the example is so small, the extra patches for journaling seem

[2]Fast File System [41], an influential file system upon which many modern file system designs are based.



a) Adding a block (soft updates) **b)** . . . plus removing a file **c)** Adding a block (journaling)

Figure 2.2: Example patch arrangements for an ext2-like file system. Circles represent patches, shaded boxes represent disk blocks, and arrows represent direct dependencies. **a)** A soft updates order for appending a zeroed-out block to a file. **b)** A different file on the same inode block is removed before the previous changes commit, inducing a circular block dependency. **c)** A journal order for appending a zeroed-out block to a file.

like a large burden. In a larger transaction, a lower proportion of extra patches would be required.

Soft updates Early file systems aimed to avoid post-crash disk inconsistencies by writing some, or all, blocks synchronously. For example, the write system call might block until all metadata writes have completed – clearly a slow interface for today's CPUs and disks. Soft updates provides post-crash consistency without synchronous writes by tracking and obeying necessary dependencies among writes. A soft updates file system orders its writes to enforce three simple rules for metadata consistency [17]:

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

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

These rules map directly to patches. Figure 2.2a shows a set of soft-updates-like patches and dependencies for our block-append operation. Soft updates Rule 1 requires that $i \rightarrow b$. Rule 2 requires that d depend on the nullification of previous pointers to the block. A simple, though more restrictive, way to accomplish this is to let $d \rightarrow b$, where b depends on any such nullifications (there are none here). The dependencies $i \rightarrow d$ and $i' \rightarrow d$ provide an additional guarantee above and beyond metadata consistency, namely that no file ever contains accessible uninitialized data.

Figure 2.2b shows how an additional file system operation can induce a circular dependency among blocks. Before the changes in Figure 2.2a commit, the user deletes a one-block file whose data block and inode happen to lie on the bitmap and inode blocks used by the previous operation. Rule 2 requires the dependency $b_2 \rightarrow i_2$, but given this dependency and the previous $i \rightarrow b$, neither the bitmap block nor the inode block can be written first! Breaking the cycle requires rolling back one or more patches, which in turn requires undo data. For example, the system might roll back b_2 and write the resulting bitmap block, which contains only b. Once this write commits, all of i, i', and i_2 are safe to write; once *they* commit, the system can write the bitmap block again, this time including b_2.

Unlike Featherstitch, the BSD UFS soft updates implementation (which has been the default consistency mechanism in BSD for over a decade) represents each UFS operation by a different specialized structure encapsulating all of that operation's disk changes and dependencies. These structures, their relationships, and their uses are quite complex [42], and involve constant micromanagement by the file system code to ensure that appropriate block data is written to disk. After each write to the disk completes, callbacks process the structures and make pending, dependent changes to other cached disk blocks. In some cases, this mechanism even resulted in userspace-visible anomalies like incorrect link counts for directories when subdirectories had been recently removed, and required the addition of new "effective" metadata fields in other kernel structures to hide them.

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

This process also maps directly to patch dependencies, as shown in Figure 2.2c. Copies of the affected blocks are written into the journal area using patches d_J, i_J, and b_J, each on its own block. Patch *cmt* creates the commit record on a fourth block in the journal area; it depends on d_J, i_J, and b_J. The changes to the main file system all depend on *cmt*. Finally, patch *cmp*, which depends on the main file system changes, overwrites the commit record with a completion record. Again, a circular block dependency requires the system to roll back a patch, namely *cmp*, and write the commit/completion block twice.

2.1.5 Patch Implementation

Our Featherstitch file system implementation creates patch structures corresponding directly to the patch abstraction we have just described. Functions like `patch_create_byte` create patches; their arguments include the relevant block, any direct dependencies, and the new data. Most patches specify this data as a contiguous byte range, including an offset into the block and the patch length in bytes. The undo data for very small patches (4 bytes or less) is stored in the patch structure itself; for larger patches, undo data is stored in separately allocated memory. In bitmap blocks, changes to individual bits in a word can have independent dependencies, which we handle with a special bit-flip patch type.

The implementation automatically detects one type of dependency. If two patches q and p affect the same block and have overlapping data ranges, and q was created after p, then Featherstitch adds an *overlap dependency* $q \rightarrow p$ to ensure that q is written after p. File systems need not detect such dependencies themselves.

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

The implementation also supports *empty* patches, which have no associated data or block. For example, during a journal transaction, changes to the main body of the disk should depend on a journal commit record that has not yet been created. Featherstitch makes these patches depend on an empty patch that is explicitly held in memory. Once the commit record is created, the empty patch is updated to depend on the actual commit record and then released. The empty patch automatically commits at the same time as the commit record, allowing the main file system changes to follow. Empty patches can shrink memory usage by representing quadratic sets of dependencies with a linear number of edges: if all m patches in Q must depend on all n patches in P, one could add an empty patch e and $m+n$ direct dependencies $q_i \rightarrow e$ and $e \rightarrow p_j$. This is useful for patchgroups; see Section 2.3. However, extensive use of empty patches adds to system time by requiring that functions traverse empty patch layers to find true dependencies. Our implementation uses empty patches infrequently, and in the rest of this section, patches are nonempty unless explicitly stated.

2.1.6 Discussion

The patch abstraction places only one substantive restriction on its users, namely, that circular dependency chains are errors. This restriction arises from the file system context: Featherstitch assumes a lower layer that commits one block at a time. Disks certainly behave this way, but a dependency tracker built above a more advanced lower layer – such as a journal – could resolve many circular dependency chains by forcing the relevant blocks into a single transaction or transaction equivalent. Featherstitch's journal module could potentially implement this, allowing upper layers to create (size-limited) circular dependency chains, but we leave not investigated this extension.

Patches model write-before relationships, but one might instead build a generalized dependency system that modeled abstract transactions. We chose write-before relationships as our foundation since they minimally constrain file system disk layout.

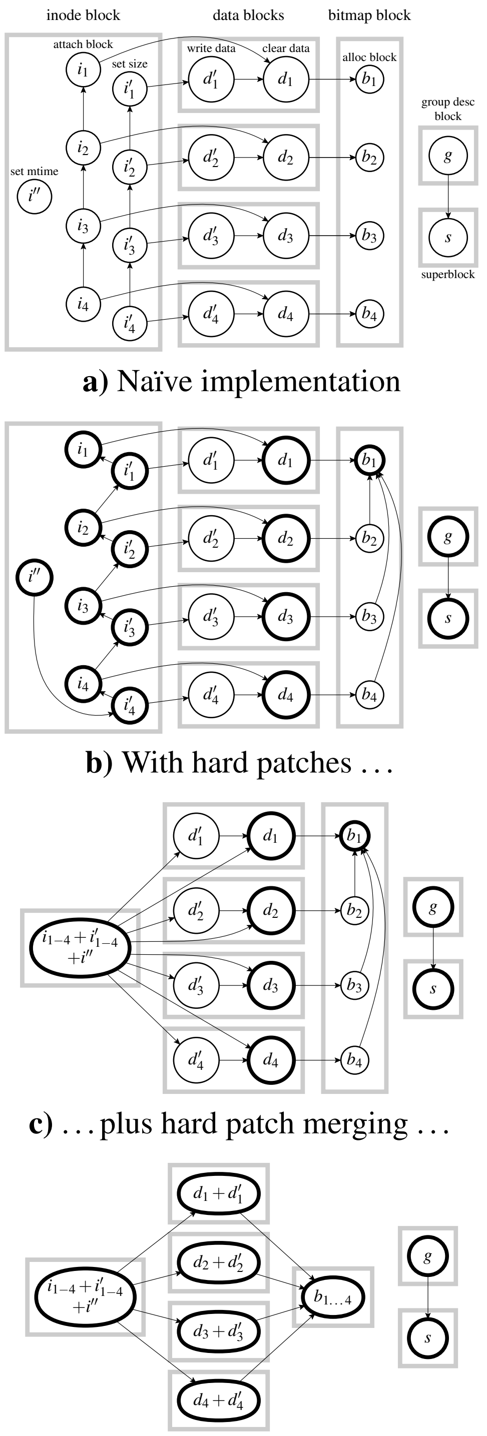
2.2 Patch Optimizations

Figure 2.3a shows the patches generated by a naïve Featherstitch implementation when an application appends 16 kiB of data to an existing empty file using four 4 kiB writes. The file system is ext2 with soft-updates-like dependencies and 4 kiB blocks. Four blocks are allocated (patches b_1–b_4), written (d_1–d_4 and d'_1–d'_4), and attached to the file's inode (i_1–i_4); the inode's file size and modification time are updated (i'_1–i'_4 and i''); and changes to the "group descriptor" and superblock account for the allocated blocks (g and s). Each application write updates the inode; note, for example, how overlap dependencies force each modification of the inode's size to depend on the previous one. A total of eight blocks are written during the operation. Unoptimized Featherstitch, however, represents the operation with 23 patches and roughly 33,000 (!) bytes of undo data. The patches slow down the buffer cache system by making graph traversals more expensive. Storing undo data for patches on data blocks is particularly painful here, since they will *never* need to be reverted. And in larger examples, the effects are even worse. For example, when 256 MiB of blocks are allocated for the application that creates 20,000 files in the untar benchmark described in Section 2.6, unoptimized Featherstitch allocates an additional 533 MiB, mostly for patches and undo data.

This section presents optimizations based on generic dependency analysis that reduce these 23 patches and 33,000 bytes of undo data to the 8 patches and 0 bytes of undo data in Figure 2.3d. Additional optimizations simplify Featherstitch's other main overhead, the CPU time required for the buffer cache to find a suitable set of patches to write. These optimizations apply transparently to any Featherstitch file system, and have dramatic effects on real benchmarks too. For instance, they reduce memory overhead in the untar benchmark from 533 MiB to just 40 MiB.

2.2.1 Hard Patches

The first optimization reduces space overhead by eliminating undo data. When a patch p is created, Featherstitch conservatively detects whether p might require reversion: that is, whether any possible future patches and dependencies could force the buffer cache to undo p before making further progress. If no future patches and dependencies could force p's reversion, then p does not need undo data, and Featherstitch does not allocate any. This makes p a **hard patch**: a patch without undo data. The system aims to reduce memory



a) Naïve implementation

b) With hard patches . . .

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

d) . . . plus overlap merging

Figure 2.3: Patches required to append 4 blocks to an existing file, without and with optimization. Hard patches are shown with heavy borders.

usage by making most patches hard. The challenge is to detect such patches without an oracle for future dependencies.

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

We now characterize one type of patch that can be made hard. We define a *block-level cycle* as a dependency chain of uncommitted patches $p_n \rightsquigarrow \cdots \rightsquigarrow p_1$ where the ends are on the same block $blk[p_n] = blk[p_1]$, and at least one patch in the middle is on a different block $blk[p_i] \neq blk[p_1]$. The patch p_n is called a *head* of the block-level cycle. Now assume that a patch $p \in \boldsymbol{U}$ is not a head of any block-level cycle. One can then show that the buffer cache can write at least one patch without rolling back p. This is trivially possible if p itself is ready to write. If it is not, then p must depend on some uncommitted patch x on a different block. However, we know that x's uncommitted dependencies, if any, are all on blocks other than p's; otherwise there would be a block-level cycle. Since Featherstitch disallows circular dependencies, every chain of dependencies starting at x has finite length, and therefore contains an uncommitted patch y whose dependencies have all committed. (If y has in-flight dependencies, simply wait for the disk controller to commit them.) Since y is not on p's block, the buffer cache can write y without rolling back p.

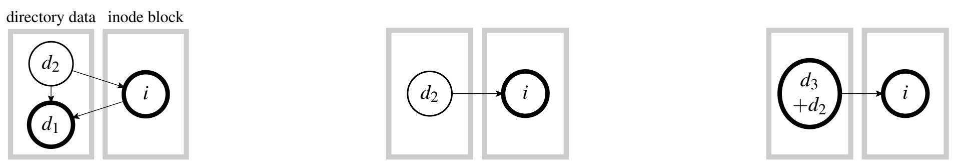
Featherstitch may thus make a patch hard when it can prove that patch will *never* be a head of a block-level cycle. Its proof strategy has two parts. First, the Featherstitch API restricts the creation of block-level cycles by restricting the creation of dependencies: *a patch's direct dependencies are all supplied at creation time*. Once p is created, the system can add new dependencies $q \rightarrow p$, but will never add new dependencies $p \rightarrow q$.[3] Since every patch follows this rule, all possible block-level cycles with head p are present in the dependency graph when p is created. Featherstitch must still check for these cycles, of course, and actual graph traversals proved expensive. We thus implemented a conservative approximation. Patch p is created as hard if *no* patches on other blocks depend on uncommitted patches on $blk[p]$ – that is, if for all $y \rightsquigarrow x$ with x an uncommitted patch on p's block, y is also on p's block. If no other block depends on p's, then clearly p can't head up a block-level cycle no matter its dependencies. This heuristic works well in practice and, given some bookkeeping, takes $O(1)$ time to check.

Applying hard patch rules to our example makes 16 of the 23 patches hard (Figure 2.3b), reducing the undo data required by slightly more than half.

2.2.2 Hard Patch Merging

File operations such as block allocations, inode updates, and directory updates create many distinct patches. Keeping track of these patches and their dependencies requires memory

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



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

Figure 2.4: Soft-to-hard patch merging. **a)** Soft-updates-like dependencies among directory data and an inode block. d_1 deletes a file whose inode is on i, so Rule 2 in §2.1.4 requires $i \rightarrow d_1$; d_2 allocates a file whose inode is on i, so Rule 1 requires $d_2 \rightarrow i$. **b)** Writing d_1 removes the cycle. **c)** d_3, which adds a hard link to d_2's file, initiates soft-to-hard merging.

and CPU time. Featherstitch therefore *merges* patches when possible, drastically reducing patch counts and memory usage, by conservatively identifying when a new patch could always be written at the same time as an existing patch. Rather than creating a new patch in this case, Featherstitch updates data and dependencies to merge the new patch into the existing one.

Two types of patch merging involve hard patches, and the first is trivial to explain: since all of a block's hard patches must be written at the same time, they can *always* be merged. Featherstitch ensures that each block contains at most one hard patch. If a new patch p could be created as hard and p's block already contains a hard patch h, then the implementation merges p into h by applying p's data to the block and setting $ddep[h] \leftarrow ddep[h] \cup ddep[p]$. This changes h's direct dependency set after h was created, but since p could have been created hard, the change cannot introduce any new block-level cycles. Unfortunately, the merge can create *intra*-block cycles: if some empty patch e has $p \rightsquigarrow e \rightsquigarrow h$, then after the merge $h \rightsquigarrow e \rightsquigarrow h$. Featherstitch detects and prunes any cyclic dependencies during the merge. Hard patch merging is able to eliminate 8 of the patches in our running example, as shown in Figure 2.3c.

Second, Featherstitch detects when a new hard patch can be merged with a block's existing *soft* patches. Block-level cycles may force a patch p to be created as soft. Once those cycles are broken (because the relevant patches commit), p could be converted to hard; but to avoid unnecessary work, Featherstitch delays the conversion, performing it only when it detects that a new patch on p's block could be created hard. Figure 2.4 demonstrates this using soft-updates-like dependencies. Consider a new hard patch h added to a block that contains some soft patch p. Since h is considered to overlap p, Featherstitch adds a direct dependency $h \rightarrow p$. Since h could be hard even including this overlap dependency, we know there are no block-level cycles with head h. But as a result, we know that there are no block-level cycles with head p, and p can be transformed into a hard patch. Featherstitch will make p hard by dropping its undo data, then merge h into p. Although this type of merging is not very common in practice, it is necessary to preserve useful invariants, such as that no hard patch has a dependency on the same block.

2.2.3 Overlap Merging

The final type of merging combines soft patches with other patches, hard or soft, when they overlap. Metadata blocks, such as bitmap blocks, inodes, and directory data, tend to accumulate many nearby and overlapping patches as the file system gradually changes; for instance, Figure 2.3's i_1–i_4 all affect the same inode field. Even data blocks can collect overlapping dependencies. Figure 2.3's data writes d'_n overlap, and therefore depend on, the initialization writes d_n – but our heuristic cannot make d'_n hard since when they are created, dependencies exist from the inode block onto d_n. Overlap merging can combine these, and many other, mergeable patches, further reducing patch and undo data overhead.

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

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

In our running example, overlap merging combines all remaining soft patches with their hard counterparts, reducing the number of patches to the minimum of 8 and the amount of undo data to the minimum of 0. In our experiments, hard patches and our patch merging optimizations reduce the amount of memory allocated for undo data in soft updates and journaling orderings by at least 99%.

2.2.4 Ready Patch Lists

A different class of optimization addresses CPU time spent in the Featherstitch buffer cache. The buffer cache's main task is to choose sets of patches P that satisfy the in-flight safety property $dep[P] \subseteq P \cup \boldsymbol{C}$. A naïve implementation would guess a set P and then traverse the dependency graph starting at P, looking for problematic dependencies. Patch merging limits the size of these traversals by reducing the number of patches. Unfortunately, even modest traversals become painfully slow when executed on every block in a large buffer cache, and in our initial implementation these traversals were a bottleneck for cache sizes above 128 blocks (!).

Luckily, much of the information required for the buffer cache to choose a set P can be precomputed. Featherstitch explicitly tracks, for each patch, how many of its direct dependencies remain uncommitted or in flight. These counts are incremented as patches are added to the system and decremented as the system receives commit notifications from the disk. When both counts reach zero, the patch is safe to write, and it is moved into a *ready list* on its containing block. The buffer cache, then, can immediately tell whether a block has writable patches. To write a block B, the buffer cache initially populates the set P with the contents of B's ready list. While moving a patch p into P, Featherstitch checks whether there exist dependencies $q \to p$ where q is also on block B. The system can write q at the same time as p, so q's counts are updated as if p has already committed. This may make q ready, after which it in turn is added to P. (This premature accounting is safe because the system won't try to write B again until p and q actually commit.)

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

2.2.5 Other Optimizations

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

$$\begin{tikzcd} s \arrow[r] \arrow[rr, bend left] \arrow[rrr, bend left] & r & q & p \end{tikzcd} \qquad \begin{tikzcd} s \arrow[r] & r \arrow[r] & q \arrow[r] & p \end{tikzcd}$$

If r, q, and p are adjacent on disk, the left-hand arrangement can be satisfied with two disk requests while the right-hand one will require four. Although neither arrangement is much harder to code, in several cases we discovered that one of our file system implementations was performing slowly because it created an arrangement like the one on the right.

Care must also be taken to avoid accidental overlap dependencies, which can occur when patches cover more bytes in a disk block than necessary. These additional dependencies enforce a chronological ordering among the overlapping patches, which would not have been required with smaller, independent patches. Patches that change one independent field at a time generally give the best results. For instance, inode blocks contain multiple inodes, and changes to two inodes should generally be independent; a similar statement holds for directories. Featherstitch will merge these patches when appropriate, but if they cannot be merged, minimal patches tend to cause fewer patch reversions and give more flexibility in write ordering.

File system implementations can also generate better dependency arrangements when they can detect that certain states will never appear on disk—in particular, when they can detect that previous changes are being undone before being written to disk. For example, soft updates requires that clearing an inode depend on nullifications of all corresponding directory entries, which normally induces dependencies from the inode onto the directory entries. However, if the file was recently created and its directory entry has yet to be written to disk, then a patch to remove the directory entry might merge with the patch that created it (which itself depends on the patch initializing the inode). In that case, there is no need for a dependency in *either* direction between the inode and directory entry blocks, because the directory entry will *never* exist on disk. Leaving out these dependencies can speed up the system by avoiding block-level cycles, such as those in Figure 2.4, and the rollbacks and double writes they cause. The Featherstitch ext2 module implements these optimizations, significantly reducing disk writes, patch allocations, and undo data required when files are created and deleted within a short time. Although the optimizations are file system specific, the file system implements them using general properties, namely, whether two patches successfully merge.

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

2.3 Patchgroups

Currently, robust applications can enforce necessary write-before relationships, and thus ensure the consistency of on-disk data even after system crash, in only limited ways: they can force synchronous writes using `sync`, `fsync`, or `sync_file_range`, or they can assume particular file system implementation semantics, such as journaling. With the patch abstraction, however, a process might specify just dependencies; the storage system could use those dependencies to implement an appropriate ordering. This approach assumes little about file system implementation semantics, but unlike synchronous writes, the storage system can still buffer, combine, and reorder disk operations.

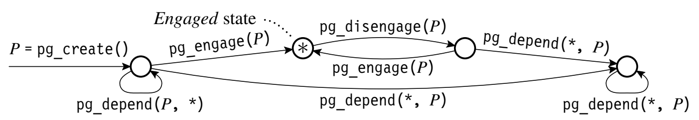


Figure 2.5: Patchgroup lifespan.

This section describes *patchgroups*, an example API for extending patches to userspace. Applications *engage* patchgroups to associate them with subsequent file system changes; dependencies are defined among patchgroups. A parent process can set up a dependency structure that its child process will obey unknowingly. Patchgroups can apply to any file system, and even raw block device writes, as they are implemented as a module within Featherstitch. Just as patches allow Featherstitch to be broken into modules, patchgroups should enable applications with specific consistency requirements to be made more modular as well.

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

2.3.1 Interface and Implementation

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

```
typedef int pg_t;
pg_t  pg_create(void);
int   pg_depend(pg_t Q, pg_t P); /* adds Q ⇝ P */
int   pg_engage(pg_t P);
int   pg_disengage(pg_t P);
int   pg_sync(pg_t P);
int   pg_close(pg_t P);
```

Each process has its own set of patchgroups. The call `pg_depend(Q, P)` makes patchgroup *Q* depend on patchgroup *P*: all patches associated with *P* will commit prior to any of those associated with *Q*. *Engaging* a patchgroup with `pg_engage` causes subsequent file system operations to be associated with that patchgroup, until it is *disengaged*. Any number of patchgroups can be engaged at once; file system operations will be associated with all currently engaged patchgroups. `pg_sync` forces an immediate write of a patchgroup to disk. `pg_create` creates a new patchgroup and returns its ID, while `pg_close` disassociates a patchgroup ID from the underlying patches which implement it.

Whereas Featherstitch modules are presumed to not create cyclic dependencies, the kernel cannot safely trust user applications to be so well behaved, so the patchgroup API makes cycles unconstructable. Figure 2.5 shows when different patchgroup dependency operations are valid. As with patches themselves, all a patchgroup's direct dependencies are added first. After this, a patchgroup becomes engaged (allowing file system operations to be associated with it) zero or more times; however, once a patchgroup *P* gains dependency via `pg_depend(*, P)`, it is sealed and can never be engaged again. This prevents applications from using patchgroups to hold dirty blocks in memory: *Q* can depend on *P* only once the system has seen the complete set of *P*'s changes.

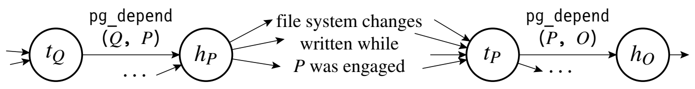


Figure 2.6: How patchgroups are implemented in terms of patches (simplified). Empty patches h_P and t_P bracket file system patches created while patchgroup P is engaged. `pg_depend` connects one patchgroup's t patch to another's h.

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

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

Without the patchgroup interface, an explicit `fsync` would be required after writing `out` (and before removing `in`) in order to achieve comparable consistency semantics. Further, it would force `out` to be written immediately, which in many cases may not be required and can hurt performance.

Patchgroups are implemented within Featherstitch by a special L2FS module. It uses several custom hooks into the rest of the kernel to be notified when processes fork and exit, and registers an `ioctl` handler on a control device with which it implements the patchgroup user interface. These process hooks are shared between all instances of the patchgroup module, so that multiple instances can be active at once. Each patchgroup corresponds to a pair of containing empty patches, and each inter-patchgroup dependency corresponds to a dependency between the empty patches. The patchgroup module inserts all file system changes made through it between the containing empty patches of any currently engaged patchgroups in the calling process. (That is, it creates incoming and outgoing dependencies between the file system changes and the containing empty patches.) Figure 2.6 shows an example patch arrangement for two patchgroups. (The actual implementation uses additional empty patches for bookkeeping.)

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

2.3.2 Case Studies

We studied the patchgroup interface by adding patchgroup support to three applications: the gzip compression utility, the Subversion version control client, and the UW IMAP mail server daemon. We chose them for their relatively simple and explicit consistency requirements; we intended to test how well patchgroups implement existing consistency mechanisms rather than create new mechanisms. One effect of this choice is that versions of these applications could attain similar consistency guarantees by running on a fully-journaled file system with a conventional API, although at least IMAP would require modification to do so. Patchgroups, however, make the required guarantees explicit, can be implemented on other types of file systems, and introduce no additional cost on fully-journaled systems.

Gzip Our modified gzip [18] uses patchgroups to make the input file's removal depend on the output file's data being written; thus, a crash cannot lose both files. The update adds 10 lines of code to gzip v1.3.9, showing that simple consistency requirements are simple to implement with patchgroups.

Subversion The Subversion version control system's client [67] manipulates a local working copy of a repository. The working copy library is designed to avoid data corruption or loss should the process exit prematurely from a working copy operation. This safety is achieved using application-level write ahead journaling, where each entry in Subversion's journal is either idempotent or atomic. Depending on the file system, however, even this precaution may not protect a working copy operation against a system crash. For example, the journal file is marked as complete by moving it from a temporary location to its live location. Should the file system completely commit the file rename before the file data, and crash before completing the file data commit, then a subsequent journal replay could corrupt the working copy.

The working copy library could ensure a safe commit ordering by syncing files as necessary, and the Subversion server (repository) library takes this approach, but developers deemed this approach too slow to be worthwhile at the client [61]. Instead, the working copy library assumes that first, all preceding writes to a file's data are committed before the file is renamed, and second, metadata updates are effectively committed in their system call order. This does not hold on many systems; for example, neither NTFS with journaling nor BSD UFS with soft updates provide the required properties. The Subversion developers essentially specialized their consistency mechanism for only ext3 and ext4.[4]

[4]As part of the delayed allocation performance feature added in ext4, the developers of the ext3 and ext4 file systems originally decided to not include the rename guarantee in the file system's default consistency mode (`data=ordered`). Providing this guarantee can significantly increase the time durability requests (e.g., `fsync`) take because the file system must write all dirty data blocks before it may commit any metadata changes. Additionally, the implementation's original purpose was only to enforce a security guarantee (to prevent deallocated data from becoming visible in other files after a crash). The ability for it to also enforce a file consistency guarantee was "an accident" [72]. As far as we are aware, the ext4 developers did not know that applications had come to depend on this behavior for application correctness.

When distributions began to support ext4, users began to report that crashes were causing data losses (non-empty files became empty) [78]. Ted Ts'o, an ext4 developer, responded with two points. First, the ext4 developers had queued patches for Linux 2.6.30 to reimplement the rename guarantee that applications had come to depend upon. Second that, even though such a `rename` will not require a prior `fsync`, applications should still change their behavior to do so [71–76]. A number of Linux Kernel Mailing List threads, many

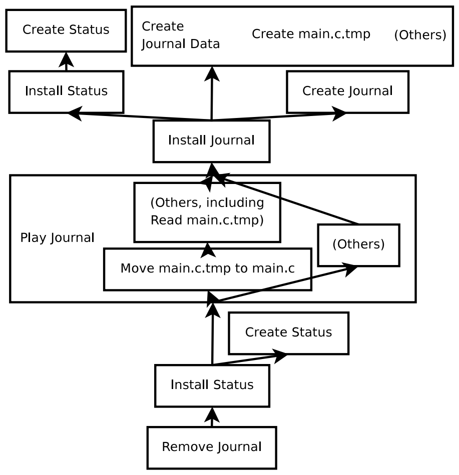


Figure 2.7: Patchgroups to update a file `main.c` in a Subversion working copy

We updated the Subversion working copy library to express commit ordering requirements directly using patchgroups. Figure 2.7 shows the patchgroups created to update a file with conflicting local and repository changes. The file rename property was replaced in two ways. Files created in a temporary location and then moved into their live location, such as directory status and journal files, now make the rename depend on the file data writes; but files only referenced by live files, such as updated file copies used by journal file entries, can live with a weaker ordering: the installation of referencing files is made to depend on the file data writes. The use of linearly ordered metadata updates was also replaced by patchgroup dependencies, and making the dependencies explicit let us reason about Subversion's actual order requirements, which are much less strict than linear ordering. For example, the updated file copies used by the journal may be committed in any order, and most journal playback operations may commit in any order. Only interacting operations, such as a file read and subsequent rename, require ordering.

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

---

bug comments and blog posts, and a Slashdot story followed on whether the file system should provide such an ordering guarantee separately from a durability guarantee and, if it should, what interfaces would work well.

Following the ext4 consistency issue coming to light, developers updated many applications to follow the advised rename practice (e.g., the GLib `g_file_set_contents` [32] and dpkg [27, 31]). However, many have also found that the added durability requests can significantly decrease performance (e.g., Ubuntu installs became slower with the dpkg change [26]). Current practice appears to be a mix of following the advised rename practice for all such renames; following it only for renames that developers deem essential for their users, because developers find that doing so for all renames is too slow; depending on the rename consistency guarantee that some file systems provide, because `fsync` requests are unnecessary on those file systems and developers deemed `fsync` too slow; and not attempting to ensure consistency.

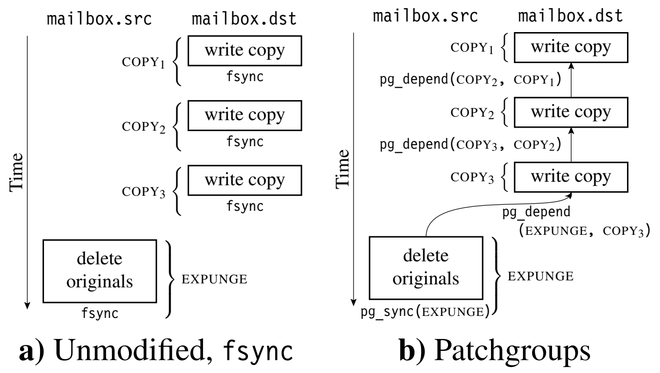


Figure 2.8: UW IMAP server, without and with patchgroups, moving three messages from `mailbox.src` to `mailbox.dst`.

UW IMAP We updated the University of Washington's IMAP mail server (v2004g) [49] to ensure mail updates are safely committed to disk. The Internet Message Access Protocol (IMAP) [10] provides remote access to a mail server's email message store. The most relevant IMAP commands synchronize changes to the server's disk (CHECK), copy a message from the selected mailbox to another mailbox (COPY), and delete messages marked for deletion (EXPUNGE).

We updated the imapd and mbox mail storage drivers to use patchgroups, ensuring that all disk writes occur in a safe ordering without enforcing a specific block write order. The original server conservatively preserved command ordering by syncing the mailbox file after each CHECK on it or COPY into it. For example, Figure 2.8a illustrates moving messages from one mailbox to another. With patchgroups, each command's file system updates are executed under a distinct patchgroup and, through the patchgroup, made to depend on the previous command's updates. This is necessary, for example, so that moving a message to another folder (accomplished by copying to the destination file and then removing from the source file) cannot lose the copied message should the server crash part way through the disk updates. The updated CHECK and EXPUNGE commands use `pg_sync` to sync all preceding disk updates. This removes the requirement that COPY sync its destination mailbox: the client's CHECK or EXPUNGE request will ensure changes are committed to disk, and the patchgroup dependencies ensure changes are committed in a safe ordering. Figure 2.8b illustrates using patches to move messages.

These changes improve UW IMAP by ensuring disk write ordering correctness and by performing disk writes more efficiently than synchronous writes. As each command's changes now depend on the preceding command's changes, it is no longer required that all code specifically ensure its changes are committed before any later, dependent command's changes. Without patchgroups, modules like the mbox driver forced a conservative disk sync protocol because ensuring safety more efficiently required additional state information, adding further complexity. The Dovecot IMAP server's source code notes this exact difficulty [11, maildir-save.c]:

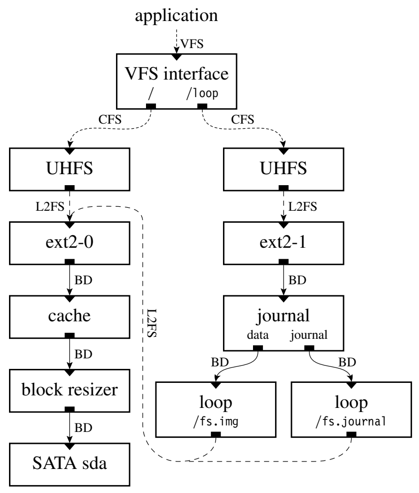


Figure 2.9: A running Featherstitch configuration. / is a soft updates file system on a SATA drive; */loop* is an externally journaled file system on loop devices.

```
/* 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 2.6.5.

2.4 Modules

Patches allow the modules in a storage system to explicitly specify what their write ordering requirements are, multiple modules can cooperate in specifying overall dependency requirements by passing patches back and forth. This allows implementers to write file system extensions, both providing and taking advantage of strong consistency guarantees, that would otherwise be difficult or impossible to implement. A typical Featherstitch configuration is composed of many such modules; Figure 2.9 shows an example configuration that exposes two file systems. Featherstitch modules fall into three major categories.

Block device (BD) modules are closest to the disk, and have a fairly conventional block device interface with interfaces such as “read block” and “flush.” For example, the module that enforces patch dependencies, the buffer cache module, is of this type. The journal module is also a block device module; it adds journaling to whatever file system is run on it by transforming the incoming dependencies.

Common file system (CFS) modules live closest to the system call interface, and have an interface similar to VFS [30]. These modules generally do not deal with patches, but can be used to implement simple “stackable” file system extensions that do not require any specific dependencies (similar to [90, 91]). For instance, Featherstitch includes a case-insensitivity module of this type.

In between these two interfaces are modules implementing a low-level file system (L2FS) interface, which helps divide file system implementations into code common across block-structured file systems and code specific to a given file system layout. The L2FS interface has functions to allocate blocks, add blocks to files, allocate file names, and other file system micro-operations. Like BD functions, L2FS functions deal with patches, allowing file system extensions that need specific write orders for consistency to be implemented as L2FS modules. A generic CFS-to-L2FS module called UHFS ("universal high-level file system") decomposes familiar VFS operations like write, read, and append into L2FS micro-operations. Our ext2, UFS, and "waffle" file system modules implement the L2FS interface, and sit "on top of" block device modules.

Modules examine and modify dependencies via patches passed to them as arguments. Every L2FS and BD function that might modify the file system takes as an argument the patch, if any, on which the modification should depend; when done, the function returns some patch corresponding to the modification itself.

Mike Mammarella is largely responsible for the design of the Featherstitch module system and wrote or contributed to all the modules, Andrew de los Reyes wrote the initial Linux block device, Lei Zhang wrote UFS, and Shant Hovsepian and Andrew Matsuoka wrote ext2. I am responsible in particular for the ext2 optimizations and the Linux and Featherstitch buffer cache integration (with Linux guidance from Steve VanDeBogart). And more generally, I contributed to the patchgroup, UHFS, loopback, journal, and Linux block device modules.

2.4.1 ext2, UFS, and waffle

Featherstitch currently has L2FS modules that implement three file system types: Linux ext2, 4.2 BSD UFS (Unix File System, the modern incarnation of the Fast File System [41]), and "waffle," which is a simple file system patterned after NetApp's WAFL [21]. The ext2 and UFS modules generate dependencies arranged according to the soft updates rules, and thus provide consistency; other dependency arrangements, like journaling, are achieved by transforming these. To the best of our knowledge, our implementation of ext2 is the first to provide soft updates consistency guarantees. Unlike FreeBSD's soft updates implementation, once these modules set up dependencies, they no longer need to concern themselves with file system consistency: the block device subsystem will track and enforce dependencies.

For the most part, these modules are merely reimplementations of the corresponding original versions in Linux and BSD, although somewhat simplified due to the UHFS module handling part of the work. The key property of these modules that distinguishes them from the original versions is that they generate patches, allowing other modules to examine, add to, and change the dependencies.

The waffle module, unlike the other two L2FS modules, generates dependencies arranged for shadow paging, where no block that is currently reachable from the file system root on disk may be written. Rather, a copy is made and updated, and all pointers to the block are updated (possibly recursively causing more blocks to be cloned). Periodically, the single root block is updated (in place) to point to the new tree of blocks, atomically switching from the old version of the file system to the new version. In this design, patch dependencies always point downwards, from the root to the leaves, so that the root can only be written once all the data to which it refers has also been written. In fact, the dependencies need not form a deep tree – the patch that updates the root block can just directly depend on all the others, giving the cache maximum flexibility in choosing which blocks to write first.

2.4.2 Journal

The journal module is a block device module that automatically makes any block device journaled. It does this by transforming the incoming patches, presumably generated by a file system module like ext2, into patches implementing journal transactions. It uses a separate journal block device to store the journal, allowing many different possible configurations. For instance, the journal can be stored on a different partition on the same disk, or on a separate disk or a network block device. The journal block device can even be a loopback block device to a file within another file system (as it is in Figure 2.9) or, to produce an "internal" journal, to the journaled file system itself. No special provisions are necessary to allow these configurations: patches convey all the required dependency information automatically. (The journal module does however have to detect recursive calls into itself, and not attempt to journal the journal blocks, when an internal journal is in use.)

Modified blocks are copied into the journal device by creating new patches. A commit record patch is also created that depends on these other journal device patches; the original patches are in turn altered to depend on the commit record. Any soft-updates-like dependencies among the original patches are removed, since they are not needed when the journal handles consistency; however, the journal does obey user-specified dependencies, in the form of patchgroups (see §2.3). Finally, a completion record, which overwrites the commit record, is created depending on the original patches. This arrangement is also described in Section 2.1.4 and depicted in Figure 2.2.

The journal format is similar to ext3's [77]: a transaction contains a list of block numbers, the data to be written to those blocks, and finally a single commit record. Although the journal modifies existing patches' direct dependencies, it ensures that any new dependencies do not introduce block-level cycles. (This is a statically worked out guarantee based on careful analysis of the code involved. It is not checked at run-time, unless specific debugging options are enabled.)

As in ext3, transactions are required to commit in sequence. The journal module sets each commit record to depend on the previous commit record, and each completion record to depend on the previous completion record. This allows multiple outstanding transactions in the journal, which benefits performance, but ensures that in the event of a crash, the journal's committed transactions will be contiguous.

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

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

We also provide several other modules that modify dependencies, including an "asynchronous mode" module that removes all dependencies, allowing the buffer cache to write blocks in any order. Using the journal or asynchronous mode modules, the same ext2 module can be used in asynchronous, soft updates, or journaled modes.

2.4.3 Buffer Cache

The Featherstitch buffer cache both caches blocks in memory and ensures that modifications are written to stable storage in a safe order. Modules "below" the buffer cache – that is, between its output interface and the disk – are considered part of the "disk controller"; they can reorder block writes at will without violating dependencies, since those block writes will contain only in-flight patches. The buffer cache sees the complex consistency mechanisms that other modules define as nothing more than sets of dependencies among patches; it has no idea what consistency mechanisms it is implementing, if any. In some sense, it is the "core" of a working Featherstitch system: it makes unnecessary the ad-hoc, fragile, and obfuscated buffer cache micromanagement required with a "dumb" buffer cache, and replaces it with generic dependency enforcement performed by a dedicated module.

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

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

2.5 Implementation

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

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

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

Our prototype has several performance problems caused by incomplete Linux integration. For example, writing a block requires copying that block's data whether or not any patches were undone, and our buffer cache currently stores all blocks in permanently-mapped kernel memory, limiting the buffer cache's maximum size.

2.6 Evaluation

We first evaluate the effectiveness of patch optimizations. Next, we evaluate the performance of Featherstitch relative to Linux ext2 and ext3 using a variety of benchmarks, including the widely-used PostMark [29] and modified Andrew file system [22] benchmarks. Then we briefly evaluate the consistency properties and general correctness of the Featherstitch implementation by forcing spontaneous crashes and examining the state of the resulting disk images. Finally, we evaluate the performance of patchgroups using an IMAP server modified to use them and a simple benchmark moving many messages. This evaluation shows that patch optimizations significantly reduce patch memory and CPU requirements; that a Featherstitch patch-based storage system has overall performance competitive with Linux, though using up to four times more CPU time; that Featherstitch file systems are consistent after system crashes; and that a patchgroup-enabled IMAP server outperforms the unmodified server on Featherstitch.

2.6.1 Methodology

All tests were run on a Dell Precision 380 with a 3.2 GHz Pentium 4 CPU (with hyper-threading disabled), 2 GiB of RAM, and a Seagate ST3320620AS 320 GB 7200 RPM SATA2 disk. Tests use a 10 GiB file system and the Linux 2.6.20.1 kernel with the Ubuntu v6.06.1 distribution. Because Featherstitch uses only permanently-mapped memory, we disable high memory for all configurations, limiting the computer to 912 MiB of RAM. Only the PostMark benchmark performs slower due to this cache size limitation. All timing results are the mean over five runs.

To evaluate patch optimizations and Featherstitch as a whole we ran four benchmarks. The *untar benchmark* untars and syncs the Linux 2.6.15 source code from the cached file `linux-2.6.15.tar` (218 MiB). The *delete benchmark*, after unmounting and remounting the file system following the untar benchmark, deletes the result of the untar benchmark and syncs. The *PostMark benchmark* emulates the small file workloads seen on email and netnews servers [29]. We use PostMark v1.5, configured to create 500 files ranging in size from 500 B to 4 MB; perform 500 transactions consisting of file reads, writes, creates, and deletes; delete its files; and finally sync. The modified *Andrew benchmark* [22] emulates a software development workload. The benchmark creates a directory hierarchy, copies a source tree, reads the extracted files, compiles the extracted files, and syncs. The source code we use for the modified Andrew benchmark is the Ion window manager, version 2-20040729.

2.6.2 Optimization Benefits

We evaluate the effectiveness of the patch optimizations discussed in Section 2.2 in terms of the total number of patches created, amount of undo data allocated, and system CPU time used. Figure 2.10 shows these results for the untar, delete, PostMark, and Andrew benchmarks for Featherstitch ext2 in soft updates mode, with all combinations of using hard patches and overlap merging. The PostMark results for no optimizations and for just the hard patches optimization use a smaller maximum Featherstitch cache size, 80,000 blocks vs. 160,000 blocks, so that the benchmark does not run our machine out of memory. Optimization effectiveness is similar for journaling configurations.

Optimization	# Patches	Undo data	System time
Untar			
None	619,740	459.41 MiB	3.33 sec
Hard patches	446,002	205.94 MiB	2.73 sec
Overlap merging	111,486	254.02 MiB	1.87 sec
Both	68,887	0.39 MiB	1.83 sec
Delete			
None	299,089	1.43 MiB	0.81 sec
Hard patches	41,113	0.91 MiB	0.24 sec
Overlap merging	54,665	0.93 MiB	0.31 sec
Both	1,800	0.00 MiB	0.15 sec
PostMark			
None	4,590,571	3,175.28 MiB	23.64 sec
Hard patches	2,544,198	1,582.94 MiB	18.62 sec
Overlap merging	550,442	1,590.27 MiB	12.88 sec
Both	675,308	0.11 MiB	11.05 sec
Andrew			
None	70,932	64.09 MiB	4.34 sec
Hard patches	50,769	36.18 MiB	4.32 sec
Overlap merging	12,449	27.90 MiB	4.20 sec
Both	10,418	0.04 MiB	4.07 sec

Figure 2.10: Effectiveness of Featherstitch optimizations.

Both optimizations work well alone, but their combination is particularly effective at reducing the amount of undo data, which, again, is pure overhead relative to conventional file systems. Undo data memory usage is reduced by at least 99%, the number of patches created is reduced by 85–99%, and system CPU time is reduced by up to 81%. These savings reduce Featherstitch memory overhead from 145–355% of the memory allocated for block data to 4–18% of that memory, a 95–97% reduction. For example, Featherstitch allocations are reduced from 3,321 MiB to 74 MiB for the PostMark benchmark, which sees 2,165 MiB of block allocations.[5]

2.6.3 Benchmarks and Linux Comparison

We benchmark Featherstitch and Linux for all four benchmarks, comparing the effects of different consistency mechanisms and comparing patch-based with non-patch-based implementations. Specifically, we examine Linux ext2 in asynchronous mode; ext3 in writeback and full journal modes; and Featherstitch ext2 in asynchronous, soft updates, metadata journal, and full journal modes. All file systems were created with default configurations, and

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

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

Figure 2.11: Benchmark times (seconds). System CPU times are in square brackets. Safe configurations are **bold**.

all journaled file systems used a 64 MiB journal. Ext3 implements three different journaling modes, which provide different consistency guarantees. The strength of these guarantees is strictly ordered as "writeback $<$ ordered $<$ full." Writeback journaling commits metadata atomically and commits data only after the corresponding metadata. Featherstitch metadata journaling is equivalent to ext3 writeback journaling. Ordered journaling commits data associated with a given transaction prior to the following transaction's metadata, and is the most commonly used ext3 journaling mode. Doing this requires ensuring that blocks allocated during a transaction were not in use prior to the transaction – otherwise, if the transaction is interrupted before it commits, the previous uses of those blocks will be clobbered. While this concern is orthogonal to the use of patches, it does require that the block allocator be aware of transactions, or that the journal module can hook into the block allocator to ensure this; Featherstitch does not currently have either of these features and so does not provide ordered mode journaling. In all tests ext3 writeback and ordered journaling modes performed similarly, and Featherstitch does not implement ordered mode, so we report only writeback results. Full journaling commits data atomically.

There is a notable write durability difference between the default Featherstitch and Linux ext2/ext3 configurations: Featherstitch safely presumes a write is durable after it is on the disk platter, whereas Linux ext2 and ext3 by default presume a write is durable once it reaches the disk cache. However, Linux can write safely, by restricting the disk to providing only a write-through cache, and Featherstitch can write unsafely by disabling FUA. We distinguish safe (FUA or a write-through cache) from unsafe results when comparing the systems. Although safe results for Featherstitch and Linux utilize different mechanisms (FUA vs. a write-through cache), we note that Featherstitch performs identically in these benchmarks when using either mechanism.

The results are listed in Figure 2.11; safe configurations are listed in bold. In general, Featherstitch performs comparably with Linux ext2/ext3 when providing similar durability guarantees. Linux ext2/ext3 sometimes outperforms Featherstitch (for the PostMark test in journaling modes), but more often Featherstitch outperforms Linux. There are several possible reasons, including slight differences in block allocation policy, but the main point is that Featherstitch's general mechanism for tracking dependencies does not significantly degrade total time. Unfortunately, Featherstitch can use up to four times more CPU time than Linux ext2 or ext3. (Featherstitch and Linux have similar system time results for the Andrew benchmark, perhaps because Andrew creates relatively few patches even in the unoptimized case.) Higher CPU requirements are an important concern and, despite much progress in our optimization efforts, remain a weakness. Some of the contributors to Featherstitch CPU usage are inherent, such as patch creation, while others are artifacts of the current implementation, such as creating a second copy of a block to write it to disk; we have not separated these categories.

2.6.4 Correctness

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

2.6.5 Patchgroups

We evaluate the performance of the patchgroup-enabled UW IMAP mail server by benchmarking moving 1,000 messages from one folder to another. To move the messages, the client selects the source mailbox (containing 1,000 2 kiB messages), creates a new mailbox, copies each message to the new mailbox and marks each source message for deletion, expunges the marked messages, commits the mailboxes, and logs out.

Figure 2.12 shows the results for safe file system configurations, reporting total time, system CPU time, and the number of disk write requests (an indicator of the number of required seeks in safe configurations). We benchmark Featherstitch and Linux with the unmodified server (sync after each operation), Featherstitch with the patchgroup-enabled server (`pg_sync` on durable operations), and Linux and Featherstitch with the server modified to assume and take advantage of fully journaled file systems (changes are effectively committed in order, so sync only on durable operations). Only safe configurations are listed; unsafe configurations complete in about 1.5 seconds on either system. Featherstitch meta and full journal modes perform similarly; we report only the full journal mode. Linux ext3 writeback, ordered, and full journal modes also perform similarly; we again report only the

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

Figure 2.12: IMAP benchmark: move 1,000 messages. System CPU times shown in square brackets. Writes are in number of requests. All configurations are safe.

full journal mode. Using an `fsync` per durable operation (CHECK and EXPUNGE) on a fully journaled file system performs similarly for Featherstitch and Linux; we report the results only for Linux full journal mode.

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

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

2.6.6 Evaluation Summary

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

2.7 Related Work

Most modern file systems protect file system integrity in the face of possible power failure or crashes via journaling, which groups operations into transactions that commit atomically [62]. The content and the layout of the journal vary in each implementation, but in all cases, the system can use the journal to replay (or roll back) any transactions that did not complete due to the shutdown. A recovery procedure, if correct [89], avoids time-consuming file system checks on post-crash reboot in favor of simple journal operations.

Soft updates [17] is another important mechanism for ensuring post-crash consistency. Carefully managed write orderings avoid the need for synchronous writes to disk or duplicate writes to a journal; only relatively harmless inconsistencies, such as leaked blocks, are allowed to appear on the file system. As in journaling, soft updates can avoid scanning the file system after a crash to detect inconsistencies, although the file system must still be scanned in the background to recover leaked storage.

Patches naturally represent both journaling and soft updates, which we use as running examples throughout this chapter. In each case, our patch implementation extracts ad hoc orderings and optimizations into general dependency graphs, making the orderings potentially easier to understand and modify. Soft updates is in some ways a more challenging test of the patch abstraction: its dependencies are more variable and harder to predict, it is widely considered difficult to implement, and the existing FreeBSD implementation is quite optimized [42]. We therefore frequently discuss soft-updates-like dependencies. This should not be construed as a wholesale endorsement of soft updates, which relies on a property (atomic block writes) that many disks do not provide, and which often requires more seeks than journaling despite writing less data.

While journaling and soft updates are the most common file system consistency mechanisms currently in use, patches were designed to represent any write-before relationship. In Section 2.4.1, we present a module that uses patches to implement shadow paging-style techniques as found in write anywhere file layouts [21]; other arrangements, like ACID transactions [87], should also be possible.

CAPFS [80] and Echo [40] considered customizable application-level consistency protocols in the context of distributed, parallel file systems. CAPFS allows application writers to design plug-ins for a parallel file store that define what actions to take before and after each client-side system call. These plug-ins can enforce additional consistency policies. Echo maintains a partial order on the locally cached updates to the remote file system, and guarantees that the server will store the updates accordingly; applications can extend the partial order. Both systems are based on the principle that not providing the right consistency protocol can cause unpredictable failures, yet enforcing unnecessary consistency protocols can be extremely expensive. Featherstitch patchgroups generalize this sort of customizable consistency and bring it to disk-based file systems.

A similar application interface to patchgroups is explored in Chapter 4 of Burnett's dissertation [7]. However, the methods used to implement the interfaces are very different: Burnett's system tracks dependencies among system calls, associates dirty blocks with unique IDs returned by those calls, and duplicates dirty blocks when necessary to preserve ordering. Featherstitch tracks individual changes to blocks internally, allowing kernel modules a finer level of control, and only chooses to expose a userspace interface similar to Burnett's as a means to simplify the sanity checking required of arbitrary user-submitted requests. Additionally, our evaluation uses a real disk rather than trace-driven simulations.

Dependencies have been used in BlueFS [47] and xsyncfs [48] to reduce the aggregate performance impact of strong consistency guarantees. Xsyncfs's *external synchrony* provides users with the same consistency guarantees as synchronous writes. Application writes are not synchronous, however. They are committed in groups using a journaling design, but additional write-before relationships are enforced on *non-file system* communication: a journal transaction must commit before output from any process involved in that transaction becomes externally visible via, for example, the terminal or a network connection. Dependency relationships are tracked across IPC as well. Featherstitch patches could be used to link file system behavior and xsyncfs process dependencies, or to define cross-network dependencies as in BlueFS; this would remove, for instance, xsyncfs's reliance on ext3. Conversely, Featherstitch applications could benefit from the combination of strict ordering and nonblocking writes provided by xsyncfs. Like xsyncfs, stackable module software for file systems [20, 60, 64, 86, 87, 90, 91] and other extensions to file system and disk interfaces [23, 63] might benefit from a patch-like mechanism that represented write-before relationships and consistency requirements agnostically.

Systems developed since Featherstitch have also realized the benefits of making the buffer cache aware of dependencies. For instance, Valor [66] provides transactional semantics at the file system level. Based on experience from previous systems, they strived to modify the kernel as little as possible; nevertheless, one of Valor's two key kernel modifications is the addition of a simple form of dependencies to the kernel's buffer cache.

Some systems have generalized a *single* consistency mechanism. Linux ext3's and ext4's journaling layers are reusable components theoretically suitable for use by any file system; however, each of ext3 and ext4 uses its own variant (JBD and JBD2, respectively) and the OCFS2 file system [51] is the only external user [34]. XN enforces a variant of soft updates on any associated library file system, but still requires that those file systems implement soft updates again themselves [28]. The FreeBSD GEOM module gjournal [13] journals writes to block devices for supported file systems, but it only provides a full journal mode.

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

2.8 Summary

Featherstitch patches provide a new way for file system implementations to formalize the “write-before” relationship among buffered changes to stable storage. Thanks to several optimizations, the performance of our prototype is in many cases at least as fast as Linux when configured to provide similar consistency guarantees. Patches simplify the implementation of consistency mechanisms like journaling and soft updates by separating the specification of write-before relationships from their enforcement. Using patches also allows our prototype to be divided into modules that cooperate loosely to implement strong consistency guarantees. The enforcement of dependencies by a dedicated module allows user applications to specify custom dependencies, via the patchgroup module, in addition to any generated within the storage system. This provides the buffer cache more freedom to reorder writes without violating the application’s needs, while simultaneously freeing the application from having to micromanage writes to disk. We present results for an IMAP server modified to take advantage of this feature, and show that it can significantly reduce both the total time and the number of writes required for our benchmark.

Chapter 3

BPFS

Chapter 2 described how patches can be used to implement any consistency mechanism and the benefits this common abstraction enables. However, Featherstitch's design assumes an underlying block-based persistent storage with high latency. What if the underlying storage technology changed – would consistency remain as qualitatively difficult to implement? This chapter describes a new consistency mechanism and new hardware that provide radically stronger guarantees than today's file systems, which are designed around the seek delays inherent with disks.

New *byte-addressable* persistent memory technologies (*BPRAM*), such as phase change memory and memristors, eliminate many of the traditional differences between volatile and non-volatile storage. These technologies are byte-addressable like DRAM, persistent like disk and flash, and up to four orders of magnitude faster than disk or flash for typical file system I/O. BPRAM can be placed side-by-side with DRAM on the memory bus, available to ordinary loads and stores by a CPU. Since BPRAM is persistent storage, the file system interface is a natural way to expose it to applications.

We have developed a new file system designed for BPRAM, called BPFS, which commits *each file system operation* synchronously and atomically. That is, BPFS commits file system operations in the order that applications execute them, it makes each commit durable before the file system operation returns to the application, and it commits each operation either completely or not at all with respect to hardware and software failures. BPFS's approach to storage differs from traditional file systems in several ways. First, BPFS file system operations write directly to persistent storage. In previous file systems, operations typically only write to volatile buffers, which the file system flushes to persistent storage every 5–150 seconds [71]. This change eliminates the window of vulnerability during which an operation has returned but its effects are not yet durable. Second, BPFS does not use a DRAM buffer cache. This both reduces the number of memory copies the file system must make and frees DRAM for other purposes. Although accessing BPRAM directly is slower than accessing a DRAM buffer cache, we believe that CPU prefetching and caching hide much of this cost. Finally, BPFS is designed to commit small, random writes to BPRAM efficiently. It was once advantageous to amortize the cost of storage transfers over a large amount of data, because of block-based transfers and device latencies and throughputs. But performing large block-based writes to BPRAM can *hinder* performance by sending unneeded traffic over the memory bus. Thus, BPFS often writes only a few bytes of data in places where a traditional disk-based file system would write kilobytes.

BPFS efficiently guarantees immediate durability and operation atomicity through optimizations of its persistent data structures and commit techniques for the capabilities and performance of BPRAM. The commit techniques are derived from shadow paging [45], which provides commit atomicity by never overwriting existing data. Instead, updates to the file system are made by copying. These block address changes are propagated through additional copy-on-writes to the root of the file system. Updating the root of the file system atomically commits the change. Our changes allow BPFS to safely omit many (and, frequently, all) of these copy-on-writes, allowing BPFS to efficiently commit operations synchronously.

These safe in-place writes are made possible by the addition of byte-addressable memory and a simple but previously-elusive primitive we add to BPRAM: *atomic 8-byte writes*. These allow BPFS to commit changes by writing a single value to BPRAM while hardware ensures that power failures and crashes cannot create a corrupted file system image. Additionally, we propose a second primitive, *epoch barriers*, to allow software to declare its ordering constraints among BPRAM writes. Within these constraints CPU caches and memory controllers may buffer and reorder writes to safely improve performance over write-through caches. We also discuss how this performance improvement comes at a trade-off of durability and consistency guarantees compared to a write-through cache.

For our evaluation, we focused on the most promising BPRAM technology, called phase change memory (PCM). Because DDR-compatible PCM is not yet available, we evaluated BPFS by comparing its write traffic with existing file systems. The results imply that BPFS can execute on PCM at least as fast as existing file systems execute on hard disk drives, even though BPFS provides stronger durability and consistency guarantees.

BPFS is joint work with Jeremy Condit, Ed Nightingale, Engin Ipek, Ben Lee, Doug Burger, and Derrick Coetzee. I started this work with my coauthors as an intern with Microsoft Research during the summer of 2008. I designed the file system, developed the Linux FUSE and Windows user-level BPFS implementations, evaluated the FUSE BPFS implementation and the effectiveness and correctness of BPFS's optimizations, observed the need for hardware atomicity and ordering support, and helped correct the multiprocessor design. My coauthors developed most of the Windows kernel evaluation (§3.6.3 and 3.6.6), including turning the user-level implementation I prototyped into a full Windows file system, designed the hardware changes, and designed most of the multiprocessor support. We published a paper in SOSP 2009 [9] on much of the work in this chapter.

This first section of this chapter describes related work. Section 3.2 then gives an overview of BPFS, Section 3.3 motivates and describes the BPFS optimizations that allow the file system to provide strong guarantees, and Section 3.4 describes the BPFS implementation. Section 3.5 discusses hardware issues and presents our atomic writes and epoch barriers. Then Section 3.6 evaluates BPFS by comparing its performance with several existing file systems, by measuring the effectiveness and correctness of our optimizations, and by measuring the effectiveness of epoch barriers. Finally, Section 3.7 describes future work and Section 3.8 concludes.

3.1 Related Work

File Systems

File systems have long been optimized for their intended medium. The Fast File System (FFS) [41], Cedar file system [19], and Sprite log-structured file system [59] are classic examples of maximizing the amount of sequential I/O in order to take advantage of the strengths of disk. Likewise, file systems such as the Journaling Flash File System (JFFS) [85] tried to optimize writes into large blocks to lessen the impact of the program/ erase cycle present in flash. We have optimized the design of BPFS for the properties of PCM, most notably making use of fast, small, random writes and no longer buffering file system data or metadata in DRAM. File systems have also been designed to take advantage of architectural features to provide robustness guarantees at lower performance costs than software-only implementations could achieve. For example, the ext4 file system uses barriers to inform SCSI and SATA disk caches of write ordering requirements, soft updates builds on atomic sector writes, and cluster file systems (e.g., VMFS) use SCSI reservations [84, Section 5.7] and/or compare-and-swap.

The file system most similar to BPFS is the WAFL [21] file system. WAFL stores the file system as a copy-on-write tree structure on disk. Whenever changes are reflected to disk, the changes “bubble up” to the root of the tree. By changing the root pointer, all of the changes are committed atomically. Because the copy-on-write procedure is quite expensive, file system changes are first stored in a log in NVRAM and only later committed, in batches, to disk. Although metadata-only journaling is typically preferred over shadow paging, shadow paging’s overheads become more attractive with the fast random writes and reads provided by BPRAM. In contrast to WAFL, BPFS places all data structures directly in BPRAM, and its data structures and commit techniques allow it to efficiently reflect changes to persistent storage individually and atomically. BPFS does not use copy-on-write to provide snapshots of previous versions, as WAFL does; its in-place updates (§3.3) would complicate this feature.

ZFS [50] and Btrfs [5] also share a number of common features with WAFL. All three use a copy-on-write mechanism to commit changes and provide snapshots. ZFS limits the size of each transaction to 128 kiB, splitting larger `write` requests into multiple transactions [53]. In contrast, BPFS guarantees that each `write` is committed atomically; the size of each BPFS transaction is bounded only by available space and BPFS fails `write` requests that do not fit in one transaction. ZFS and Btrfs use checksums to detect file system corruption proactively, whereas BPFS relies on ECC to detect hardware failures.

While BPFS uses partial copy-on-writes to make arbitrarily large and complicated updates, it is often able to use only in-place writes to atomically commit updates that are small in size and/or that modify multiple locations. We exploit BPFS file system invariants to enable these techniques, an approach used at least as early as soft updates [17]. Section 2.1.4 describes the rules that soft updates follows to safely make updates in place. BPFS improves upon soft updates in two ways. First, BPFS provides stronger consistency and durability guarantees: soft updates only ensures that metadata updates do not cause the loss of metadata/data and do not corrupt the file system. Second, soft updates provides its consistency guarantees only if block writes (which are typically 1–4 kiB in size) are guaranteed to be atomic. In contrast, BPFS only requires 8 B atomic writes, and we show how this guarantee is easily provided by PCM.

Chapter 2 describes a new architecture for constructing file systems that simplifies consistency enforcement by providing a generalized dependency abstraction. We did not implement BPFS as a Featherstitch file system because Featherstitch simplifies consistency enforcement for storage systems that buffer writes in software. While evicting writes from the buffer cache of such systems can require respecting complicated dependency requirements, we have found that ordering requirements are typically simple to understand at the point that a modification is made, when details about the operation and context are readily available. Thus BPFS's elimination of the DRAM buffer cache also simplifies consistency enforcement. BPFS can write to BPRAM through a write-back cache with epoch barriers. Similarly to how a Featherstitch file system specifies ordering requirements to the buffer cache through patches, in this mode, BPFS specifies ordering requirements to the CPU caches through epoch barriers.

Consistency and Durability

In general, transaction processing systems have focused on using write-ahead logging or shadow paging to ensure the ACID properties for transactions [46]. BPFS focuses on shadow paging to maintain these properties, and can enforce the ACID properties for each file system call (with a few caveats for modification times) using a combination of atomic writes, optional epoch barriers, and conventional file system locking.

BPFS improves durability guarantees by taking advantage of a high-throughput, low latency connection to BPRAM. However, writing to BPRAM is sill slower than writing to DRAM. External synchrony [48] hides most of the costs of synchronous disk I/O by buffering user-visible outputs until all relevant disk writes have been safely committed. We view this work as complementary to our own; as long as non-volatile storage is slower than volatile storage, then external synchrony can be used to hide the costs of synchronous I/O.

Non-Volatile Memory

Other storage systems have considered the impact of non-volatile memories. eNVy [88] presented a storage system that placed flash memory on the memory bus by using a special controller equipped with a battery-backed SRAM buffer to hide the block-addressable nature of flash. With PCM, we have a memory technology that is naturally suited for the memory bus, and we investigate ways to build more efficient file systems on top of this memory.

More recently, Mogul et al. [44] have investigated operating system support for placing either flash or PCM on the memory bus alongside DRAM. They describe several policies that could be used to anticipate future data use patterns and then migrate data between fast DRAM and slow non-volatile memory appropriately.

The Rio file cache [36] took a different approach to non-volatile memory by using battery-backed DRAM to store the buffer cache, eliminating any need to flush dirty data to disk. Rio also uses a simple form of shadow paging to provide atomic metadata writes. In contrast, BPFS does away with the buffer cache entirely, building a file system directly in BPRAM. Whereas Rio provides atomicity only for small metadata updates, BPFS guarantees that arbitrarily large data and metadata updates are committed synchronously and atomically.

In the same vein as Rio, the Conquest file system [81] used battery-backed DRAM to store small files and metadata as a way of transitioning from disk to persistent RAM. In contrast, BPFS is designed to store both small and large files in BPRAM, and it uses the properties of BPRAM to achieve strong consistency and durability guarantees. Conquest's approach may be useful in conjunction with BPFS in order to use higher-capacity storage.

In general, battery-backed DRAM (BBDRAM) represents an alternative to using BP-RAM. Most of the work described in this chapter would also apply to a file system designed for BBDRAM—in particular, we would likely design a similar file system, and we could take advantage of the same hardware features. However, there are two main advantages that BPRAM has over BBDRAM. First, BBDRAM is vulnerable to correlated failures; for example, the UPS battery will often fail either before or along with primary power, leaving no time to copy data out of DRAM. Second, BPRAM density is expected to scale much better that DRAM, making it a better long-term option for persistent storage [57].

Finally, several papers have explored the use of PCM as a scalable replacement for DRAM [33, 56, 92] as well as possible wear-leveling strategies [56, 92]. This work largely ignores the non-volatility aspect of PCM, focusing instead on its ability to scale much better than existing memory technologies such as DRAM or flash. Our work focuses on non-volatility, providing novel software applications and hardware modifications that support non-volatile aspects of BPRAM.

3.2 Overview

In this section we discuss our goals for a BPRAM-based file system, the high-level design principles that guided our work, and the basic design of the BPFS file system.

3.2.1 Goal

Most storage systems ensure consistency by using journaling or shadow paging to write to persistent storage. The use of these mechanisms in file systems provides only limited consistency and durability guarantees, which complicates the design and implementation of applications that want to in turn provide guarantees to users and other applications. For example, most file systems use journaling to guarantee that metadata operations commit atomically and in order, but they do not also journal file data.

The goal of the work in this chapter is improve the guarantees that file systems can effectively provide to applications. We would like file systems to be able to guarantee that file system operations are committed *atomically*, *in program order*, and *before* the file system operation returns to the application.

We show how a file system can provide these consistency and durability improvements while simultaneously providing reasonable performance. We do so through changes to hardware and software. On the hardware side, we propose placing byte-addressable, persistent memory hardware on the memory bus and adding an atomic write primitive to the memory. On the software side, we develop file system consistency techniques that exploit these hardware properties to avoid the classes of overheads that existing techniques incur.

3.2.2 Design Principles

Now we discuss in detail three design principles that guided this work.

Expose BPRAM Directly to the CPU

Persistent storage has traditionally resided behind both a bus controller and a storage controller. Since the latency of a read or a write is dominated by the access to the device, the overhead of this architecture does not materially affect performance. Even the fastest NAND flash SSDs have latencies in the tens of microseconds, which dwarf the cost of peripheral bus accesses. (E.g., the Intel X25-E is rated at 75 and 85 microsecond read and write latencies [24].)

In contrast, technologies such as phase change memory have access latencies in the hundreds of nanoseconds [12, 33], which is only 2–5 times slower than DRAM; thus, keeping BPRAM storage technologies behind a peripheral bus would waste the performance benefits of the storage medium. Further, peripheral buses prevent software from using byte addressability because they only support block-based accesses or are only efficient for large transfers.

Thus, we propose that BPRAM be placed directly on the memory bus, side-by-side with DRAM. The 64-bit physical address space will be divided between volatile and non-volatile memory, so the CPU can directly address BPRAM with common loads and stores. This architecture keeps access latency low and allows us to take advantage of BPRAM's byte addressability, which would not be possible if BPRAM were placed on a peripheral bus or treated as another level of the memory hierarchy behind DRAM. In addition, making

BPRAM addressable permits us to use the cache hierarchy to improve the performance of persistent memory accesses.

There are three disadvantages to placing BPRAM on the memory bus. First, there is the possibility that traffic to BPRAM will interfere with volatile memory accesses and harm overall system performance; however, interference has not been observed in microarchitectural simulation [9]. Second, the amount of BPRAM available in a system is limited by BPRAM densities and the number of free DIMM slots in a machine. However, since DRAM and PCM have similar capacities at the same technology node [12, 33] we expect to have 32 GiB PCM DIMMs at the 45 nm node, which is comparable to the size of first-generation SSDs. Third, placing persistent storage on the memory bus may make it more vulnerable to stray writes. Previous work on the Rio file cache found that 1.5% of crashes with Rio storage caused corruption, as opposed to 1.1% with disk [8]. This is a weakness of making persistent storage more directly accessible.

Note that we do not propose completely replacing DRAM with BPRAM. Since BPRAM is still slower than DRAM by a factor of 2–5, and since phase change memory cells wear out after about 10^8 writes, it is still better to use DRAM for volatile and frequently-accessed data such as the stack and the heap.

Replace the DRAM Buffer Cache with CPU Caches

The file system buffer cache stores previously read data to improve performance. Caching reads avoids duplicate media reads when future requests can be serviced from the cache. Additionally, the cache allows the file system to batch writes, amortizing media and consistency technique overheads and coalescing repeated writes. However, when the performance of DRAM and storage are similar, this “optimization” can hurt file system performance. That is, it can decrease throughput and increase latency. In such cases, the buffer cache effectively adds the overhead of an additional write for each file system write.

For this reason we propose eliminating the DRAM-based buffer cache for file systems using BPRAM. We make this proposal only because the relative speed of persistent storage is now near that of volatile memory. We also propose a replacement: use the CPU caches in place of DRAM. The performance of these caches is much greater than that of BPRAM, and we believe they can offer similar benefits as DRAM buffer caches offer for disk- and flash-based file systems. In Section 3.5.1 we discuss why it might make sense to change this choice.

Provide Atomicity and Ordering in Hardware

Because a software or hardware fault may halt execution at any time, a file system must ensure that its persistent storage is internally consistent at all times. Maintaining this invariant is complicated by two issues.

One is the lack of a simple but previously-elusive primitive: writes to persistent storage that are atomic with respect to power failures. File systems typically use a combination of checksums and duplicated writes to provide atomicity without direct hardware support.

For example, the ext4 file system defaults to journaling metadata changes and storing a checksum with each journal transaction [54]. Like disks, current memory buses lack write primitives with atomic, all-or-nothing behavior relative to power failures. (For DRAM, such a primitive would be pointless.) BPRAM file systems could use checksums and/or duplicated writes to work around this limitation. Instead, however, we recommend a simple atomic memory write primitive implemented in hardware. In Section 3.5.3 we discuss how implementing failure atomicity requires only small changes to BPRAM.

The second complication is that a file system must reason about the order in which writes become durable to be able to provide durability and consistency. Cache hierarchies and memory controllers typically do not permit this in their default modes: they buffer and reorder memory writes to improve write throughput and latency. This is a sensible trade-off for writes to volatile memory. Software typically has two approaches available that allow it to reason about the order in which writes are made to DIMMs. One is to disable write buffering for the BPRAM address range (e.g., by setting the caches to write-through mode for BPRAM addresses), so that the caches and memory controller preserve the order of persistent writes. This is a mode that we use with BPFS. It is simple for software to use safely, and it is available in current hardware.

At the same time, we would like to more closely approach the performance that hardware provides when it may buffer and reorder writes. A second approach is to allow write buffering and have software periodically flush previous writes; software would use these flushes to enforce the ordering constraints that software requires for consistency. However, we believe that the additional DRAM flushes this would force and the cache-line tracking it would require of software would result in worse, not better, performance than with buffering disabled.

Therefore, we propose a third approach: a hardware mechanism for software to declare its ordering constraints to hardware. In our proposal software can issue a special write barrier that will delimit the set of previous writes from the set that follows. Each set of writes is an *epoch*, and they are separated by an *epoch barrier*. Hardware will guarantee that the epochs are written back to main memory in order, but is allowed to reorder writes within an epoch. This approach decouples ordering from durability; whereas previous approaches enforced ordering by simply flushing dirty buffers, our approach allows hardware to enforce ordering while still leaving dirty data in the cache. Our proposal requires relatively simple hardware modifications and it provides a powerful primitive with which we can build efficient, robust software. However, by buffering writes it limits the durability guarantees software can provide, and by allowing cores to issue epochs independently, it limits the consistency guarantees software can provide. For these reasons we consider both write-through and epoch-based caching modes in our work.

3.2.3 Design Basics

Here we present the basics of the design of BPFS. The following two sections describe changes to this design to make synchronous writes performant and give an overall description of BPFS.

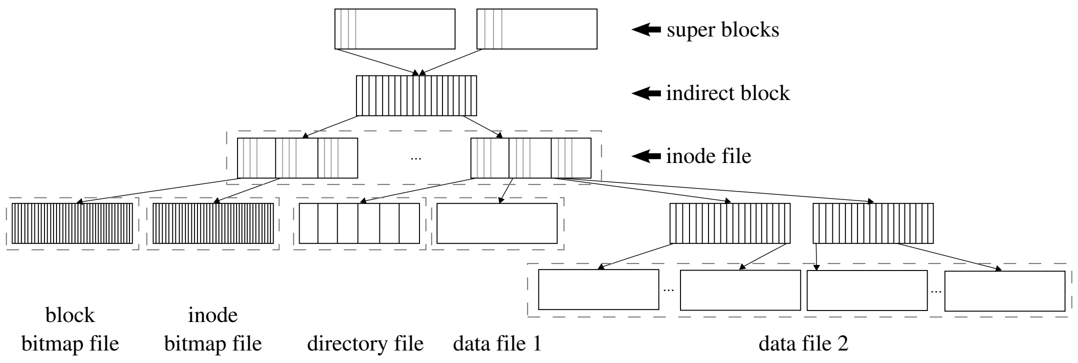


Figure 3.1: Sample shadow paging file system.

BPFS places all file system data and metadata in a tree data structure stored in persistent storage. BPFS updates this state using shadow paging to ensure the file system is consistent at all times. Its basic design and consistency mechanism are similar to the NetApp WAFL file system [21] (except that it does not provide snapshots of previous file system states). BPFS differs from typical file systems in that it commits each operation to the file system synchronously, so that when the operation returns its effects are guaranteed to be durable. Existing file systems typically either batch the commit of a set of operations, weakening durability guarantees, or log data to temporary NVRAM and later batch the recent commits to a second store, writing data twice.

Figure 3.1 shows an example file system image. The arrows in the diagram represent pointers from a parent to a child block; these form the tree through which all persistent data structures are accessible. The root of the file system is a single file, which contains an array of fixed-size inodes. These inodes contain the metadata for the remaining files. Each file consists of one to three parts. The inode stores the metadata for a file, including the addresses of the first few data blocks and the number of bytes in the file. When a file contains more than a few blocks of data the inode instead points at indirect blocks, which each contain an array of block pointers. The leaves of each tree are the blocks that store the file's data. We draw a dashed box around each set of data blocks. For a regular file, these contain the application's data. For a directory file, these contain the directory entries, which each contain a file name and reference an inode. We show each directory entry as a sub-block in the directory leaf blocks. For the inode file, these leaf blocks contain the inodes for all files except for the special case of its own inode, which BPFS stores in a separate, fixed location. We show each inode as a sub-block in the inode file leaf blocks. Finally, there are two other files internal to the file system; these are the two maps that record the allocation status for blocks and for inodes.

With shadow paging, the file system never overwrites an in-use block. Instead, it leaves the original block as is and allocates a new block for the changed data. It then propagates the address change up the tree of block references by changing each parent block, following the same copy-on-write procedure. Finally, it writes the updated reference to the file system root to the superblocks. One way to make this write atomic is to include the block's checksum inside the block and store two copies of the block. The superblock update atomically commits the changes.

Of the techniques commonly used to make file systems consistent, we chose shadow paging as a basis for BPFS because it can commit many file system changes while writing new data exactly once (rather than twice, as in journaling). Additionally, the copy-on-write primitive used in shadow paging can hide large file system changes until they are complete (as opposed to, say, soft updates, which overwrites file data in place, potentially risking visibility for intermediate states). The disadvantage of shadow paging is its copy-on-write overhead. Existing shadow paging file systems attempt to amortize these costs by batching commits. In contrast, BPFS exacerbates the overhead of copy-on-writes by committing each file system operation individually. However, in the following section we show that the inherent properties of BPRAM, and natural extensions thereto, can be used to mitigate this overhead.

3.3 Shadow Paging Optimizations

To compile Apache, as described in Section 3.6.2, the ext3 and Btrfs file systems write 65 MiB in their default modes. But to commit each file system operation synchronously, ext3 writes 700 MiB, Btrfs writes 1,200 MiB, and unoptimized BPFS would write more than 1,400 MiB. Committing each file system operation synchronously requires a shadow paging file system to immediately propagate each operation to the root of the file system. For many operations, these copy-on-writes write many more bytes than make up the direct change. Synchronous commits also preclude the file system from collapsing a series of writes and incur the overhead of performing a commit for each operation. Although our goal is to try to use BPRAM to provide stronger durability and consistency guarantees, these write traffic overheads are far larger than the performance increases we expect from BPRAM. They would result in BPFS on BPRAM being slower than existing file systems in their default modes on disk and flash.

In this section we present several optimizations that reduce this overhead. We optimize the file system's persistent data structures and commit techniques to take advantage of the byte-addressability of BPRAM, its fast random access performance, and our atomic write primitive.[1] These optimizations reduce overheads by localizing or altogether eliminating the copy-on-writes shadow paging would perform to commit. For the Apache compile benchmark, they reduce the number of bytes BPFS writes from more than 1,400 MiB to just 85 MiB.

3.3.1 Short-Circuit Commits

Propagating block address changes from each modified block to the root of the file system is a significant source of overhead for shadow paging; the sum of these copies can dwarf

[1]These optimizations would complicate snapshots.

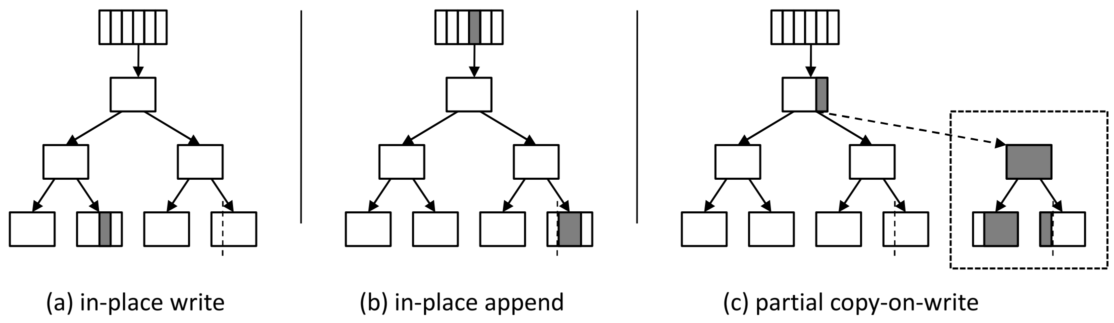


Figure 3.2: Three approaches to updating a BPFS file. Gray boxes indicate the portions of the data structures that have been updated.

the size of the desired change for small or sparse changes. The addition of atomic writes allows the file system to safely stop, or *short-circuit*, this propagation when it reaches a single write that is an ancestor of all the modified blocks. Figure 3.2a shows the simplest example of the application of this optimization. Here the file system needs to modify an aligned 8-byte region. To commit this change shadow paging would copy-on-write the containing block and then propagate the updated block number, via copy-on-writes, to the root of the file system. In contrast, short-circuiting allows the file system to safely commit this change with a single, atomic, in-place write. (This optimization is possible because of the atomic 8-byte write primitive we add to PCM; disks lack any such primitive, and thus copy-on-write to the root.) Figure 3.2c shows a more complicated commit. Here the file system needs to modify a large number of bytes in two blocks. It first copy-on-writes the two blocks. Then it modifies the two 8 byte block pointers in the parent indirect block, via a third copy-on-write. Short-circuiting allows the file system to then atomically commit the change with a single in-place write to update the grandparent indirect block. Without this optimization shadow paging would continue to copy-on-write to the root of the file system.

Alone, this optimization has limited applicability. In the following optimizations we describe how to change the file system's data structures and commit techniques to significantly increase the applicability of short-circuiting.

3.3.2 Normalized Data Structures

File systems often store resource information in multiple encodings to improve performance. For example, consider block allocation. The block pointers in inodes and indirect blocks record which blocks are allocated and which are free. In addition to these pointers, file systems often also store allocation state in an additional data structure—a block bitmap, for example. File systems do this to reduce the number of seeks the disk makes and the amount of metadata the CPU processes to find a free block for allocation. The trade-off is an increase in persistent storage space overhead and in the number of blocks the file system must commit to allocate a block. When the file system batches many allocations and/or deallocations in a commit, the amortized cost of maintaining this additional data structure is typically small. When each file system operation commits individually, as is our goal, this amortization disappears.

Further, maintaining this derived data structure limits the applicability of short-circuit commits. A file system operation that modifies block allocations must atomically commit changes to both the target file and the block bitmap file. This requires the file system to propagate copy-on-writes for the two files to their first common ancestor, likely the root. In contrast, without the block bitmap the file system would short-circuit the copy-on-write propagation while within the target file. In other words, this would reduce a sparse copy-on-write of a broad cross-section to a dense copy-on-write of a localized area.

In the following paragraphs we describe how we normalize persistent data structures to localize the modifications required to commit a file system operation.

Block and inode allocation maps BPFS avoids the broad commits that block and inode allocation maps force by doing away with these two persistent data structures. Instead of using these data structures to record resource allocation, we consider a block or inode allocated exactly when there exists a reference to it. Specifically, an inode is considered allocated when an allocated directory entry refers to it, and a block is considered allocated when a block pointer in an allocated inode refers to it. This allows the file system to allocate or free a block or inode with only localized updates, thereby requiring fewer copy-on-writes.

Although omitting these persistent maps reduces copy-on-writes, locating a free block or inode without them could be an inefficient operation; each query would require a scan of the entire file system. Instead, BPFS maintains *volatile* block and inode bitmaps. It constructs these each time the file system is mounted, using a quick and single scan of the file system. This differs from the lengthy process of a hard disk drive file system check (`fsck`) in two ways. Most significantly, with BPRAM, non-contiguous reads (seeks) have little performance overhead. Additionally, the scan only loads metadata, rather than checking and repairing the consistency of the file system data structures. In our tests with full 20 GiB file systems, scans complete in less than a second.

Inode link count Directory and file inodes can be referenced by multiple directory entries. For example, each directory contains a directory entry with the name ".." as an alias for that directory's parent. A file inode can appear under multiple names; each name sees updates to any of the others. Creating a link to an existing inode is called creating a hard link. File systems typically store the number of links to an inode in the inode data structure. This allows the file system to quickly determine when an `unlink` operation removes an inode's last link, at which point the file system will free the inode and the blocks it references. However, as with allocation maps, storing the link count in two sets of data structures forces an operation that modifies a link count to make a sparse copy-on-write of a broad cross-section of the file system. For example, to create a second link to a file, the file system must atomically write the target directory entry in its parent directory and increment the link count in the target inode. This requires the file system to propagate copy-on-writes for the two files to their first common ancestor.

We can localize these copy-on-write operations using the previous technique, for allocation maps. However, the link count data structure is less likely to fit entirely in volatile memory than an allocation map, because it records counts rather than binary presence. Additionally, the amount of space it occupies is proportional to the number of files on the persistent store. This makes it more likely to fit in persistent storage than in volatile memory, which is decoupled from the amount of persistent storage. For these reasons, we store link counts in BPRAM, not DRAM, but nevertheless treat them as volatile data. Should writes to BPRAM halt prematurely, the link count data structure may be inconsistent with the directory entries in the file system. When the file system is next mounted it can scan the directory entries to recompute link counts. This scan is similar to the inode allocation map scan. Additionally, the file system may avoid this crawl when mounting a file system that was cleanly unmounted. Although we store allocation bitmaps in DRAM, one should presumably instead store them in BPRAM, and we expect the benefits would apply similarly.

Parent directory entry File systems typically record both directions of directory parent-child relationships. That is, each directory records the list of its children while each child directory also records its parent. Storing this relationship in two forms forces operations that change the relationship to make a sparse copy-on-write of a broad cross-section of the file system. To normalize this information we eliminate the ".." directory entries for persistent storage. These directory entries are still expected by applications as they read and traverse directories, so BPFS tracks a directory's parent in its cached directory entry in volatile memory.

3.3.3 Atomic Operations with Multiple Commits

The combination of short-circuiting and normalized data structures allows the file system to commit some operations in place and many others using only localized copy-on-writes. We now describe how to further localize copy-on-write commits. We do so by reasoning about the invariants for persistent data structures to safely permit multiple commits during an atomic file system operation. In conjunction with the time optimization in the next subsection, this eliminates all copy-on-writes for several common file system operations.

In-place append As we have described, a shadow paging file system performs a copy-on-write to modify an allocated block. Writes to unallocated blocks may be made in place because the file system ignores the contents of these blocks. In-place append can be seen as an extension of this behavior to finer-grained allocations, from unallocated blocks within a file system to unallocated bytes within an allocated block. The size field in each inode records how many bytes are allocated to the file. Data in blocks allocated to the file are valid only up to the size specified in the inode; other allocated space is ignored. Consider the case of appending a few bytes to a file. When appending the new bytes does not require allocating an additional block, the file system may first write these in place and then update the size field in the inode with a second in-place write. Updating the size field atomically commits the append. Figure 3.2b illustrates this case. More generally, this optimization also allows the file system to append additional block pointers to an allocated indirect block; block pointers beyond the file size are ignored on reboot, leaving those blocks effectively free. BPFS grows a file (adding zeros) the same way an application appends zeros, except that BPFS can efficiently represent sparse blocks with null block pointers. BPFS shrinks a file by updating the size of the file with a single in-place write. Recording the block deallocations a shrink may make only needs to update the volatile block bitmap. This optimization allows the file system to change the size of a file with only in-place writes for most cases.

In-place tree height switch Even with the in-place append optimization, changing the height of a file's tree still requires a copy-on-write. This act requires a copy-on-write because the file system must atomically change two inode fields, the block pointer and the size of the file. These two fields must be committed together because the height of the tree is derived from the value of the size field. We eliminate this copy-on-write requirement by adding a new field to the inode structure for the precise purpose of allowing a third commit during the append operation. Specifically, we add a tree height field to the inode and store it in the same 8 bytes as the root block pointer. Storing the tree's block pointer and height in a single 8 byte field allows the file system to atomically update the two together with a single, in-place write. This allows BPFS to switch trees with a commit that is independent of the file size change. (An equivalent approach would be to move both fields from the inode field into their own data structure.) A crash between switching trees and setting the new file size will leave the tree taller than necessary; BPFS can recover this unnecessary space by checking each allocated inode and, when appropriate, decreasing its tree height as part of the inode crawl that also recomputes link counts after an unclean unmount. With these and the later time optimization, all file appends and truncates commit with only in-place writes.

In-place file creation To create a file, BPFS allocates and writes to an inode and a directory entry. In some cases the two above append optimizations allow this operation to commit with no copy-on-writes (when a new block is allocated for both the inode and directory entry). However, in the common case the file system must atomically modify both structures; this requires a sparse copy-on-write of a broad cross-section of the file system. We extend the in-place append optimization from arbitrary files, where the contents of the bytes allocated to a file are opaque to the file system, to metadata files, where we can leverage even finer-grained allocations. Before an inode is referenced by a directory entry, the file system can ignore its contents. Therefore BPFS may first initialize the inode in place and then reference it from the new directory entry, avoiding copy-on-writes. The same optimization applies to directory entries. Because a directory entry is considered allocated when its inode number field is not null, BPFS may first initialize the other directory entry fields and then set the inode number field with a second in-place write. Setting the inode number atomically commits the operation. This optimization and the later time optimization eliminate all copy-on-writes for file and directory creates.

The multi-commit optimizations were partially inspired by soft updates [17], which can also atomically commit some file system operations using a sequence of in-place writes. Additionally, the multi-commit optimizations follow the three soft update ordering constraints we discussed in Section 2.1.4. The guarantees that soft updates provides are weaker, however, because it buffers and reorders the effects of file system operations, does not commit all types of file system operations atomically (i.e., `write` and `rename`), and presumes that disk block writes are atomic.

3.3.4 Miscellaneous

BPFS employs two additional techniques which do not fall into the previous categories, but which significantly reduce the number of copy-on-writes the file system performs.

Independent time update As part of each file system data and metadata operation, the file system also updates the last modified or changed time that is recorded in the inode field. Including these updates in the atomic commit of the file system operation can increase the breadth of the cross-section of the file system that must be modified. In many cases, this would force BPFS to switch to copy-on-writes where none would otherwise be required. Additionally, we believe that applications typically do not depend on this guarantee. For example, in their default modes, most of today's file systems, including NTFS, HFS+, and ext3, do not atomically update a file's data and write time. For these reasons we chose to commit time updates independently from file system operations in BPFS. Although this optimization allows the file system to commit an operation using multiple commits, it differs from the earlier, atomic multi-commit optimizations because it does not preserve the atomicity of the operation. An alternative to this trade-off is to commit change and modification time updates before committing the rest of the file system operation; this would guarantee that an unmodified timestamp implies the file is unmodified. Another alternative is to log the timestamp commits and the final file system operation commit to a journal; this would atomically commit the entire operation while writing only a few additional bytes.

Packed user and group ownership The `chown` system call changes two ownership fields in an inode: the ID of the user and the ID of the group who own it. To allow short-circuiting to commit this operation with only an in-place write, we lay out the inode data structure so that these two fields are located within, and aligned to, an 8-byte region. Although this is a simple and specific change, the `chown` operation is used frequently in many workloads.

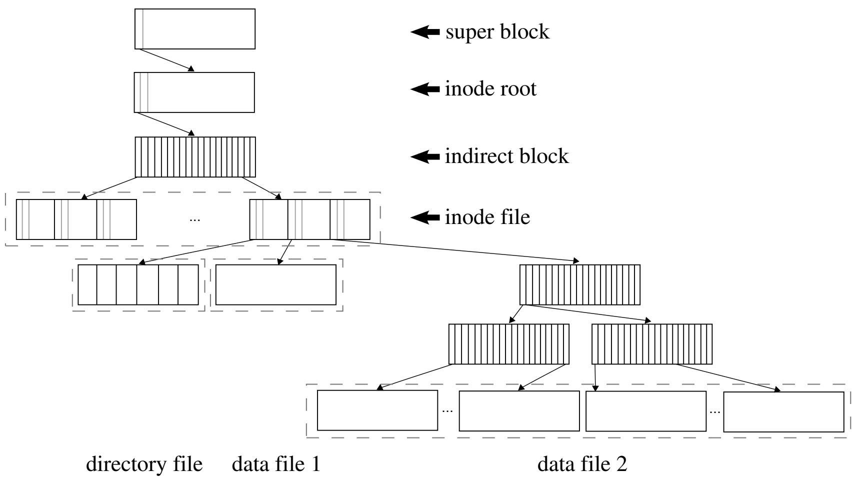


Figure 3.3: Sample BPFS file system.

3.4 Implementation

This section describes the implementation of BPFS, including its persistent and volatile data structures, software design choices, use of epoch barriers, and limitations.

3.4.1 Persistent Data Structures

Here we update and extend our description of the persistent data structures we introduced in Section 3.2.3. Figure 3.3 shows the updated version of the sample file system image from Figure 3.1.

BPFS's persistent data structures are organized into a simple tree of fixed-size (4 kiB) blocks. Block pointers are aligned 64 bit values. We chose to not store more complex data structures in BPRAM (e.g., variable block sizes or multiple pointers to a given piece of data) because we found that restricting them so has two important advantages. First, having exactly one block pointer path from the root to each block in the file system makes file system updates more efficient. Because of this simplification, copy-on-writes modify fewer leaf blocks, affect narrower cross-sections, and can often be entirely avoided. Second, fixed-sized blocks make allocation and deallocation simple.

A BPFS inode contains a single block pointer. In contrast, existing file systems often store a small array of block pointers in the inode. They do this, in part, to reduce the number of seeks needed to access data blocks; this is unimportant for BPRAM. But they also do this to reduce the amount of storage required for small files, because the file system need not allocate an indirect block for any file with less than a handful of data blocks. We omit this space optimization to instead allow BPFS to increase and decrease the height of the file's data tree with only in-place writes. Figure 3.4 shows the persistent BPFS inode data structure. In BPFS, an inode is considered valid when it is referred by a valid directory entry. Directory entries are considered valid when they contain a non-zero inode number.

```
struct height_addr
{
    uint64_t height : 3;
    uint64_t addr   : 61;
};

struct bpfs_tree_root
{
    struct height_addr ha;
    uint64_t nbytes;
};

struct bpfs_inode
{
    uint64_t generation;
    uint32_t uid;
    uint32_t gid;
    uint32_t mode;
    uint32_t nlinks; // valid at mount iff clean unmount
    uint64_t flags;
    struct bpfs_tree_root root;
    uint32_t atime;
    uint32_t ctime;
    uint32_t mtime;
    uint8_t  pad[68];
};
```

Figure 3.4: BPFS Inode.

These uses of directory entry inode numbers and block pointers replaces the inode and block bitmaps.

The height of each tree data structure is stored with the tree's root pointer. This allows the file system to update the layout of the tree and number of bytes in the tree independently.

The block pointer type has a special null value that indicates that it does not point to any block, but that its leaves are filled with zeros. This enables efficient sparse-file representation at all points in the tree. The file size can be larger or smaller than the amount of data represented by the tree itself. If it is larger, then the tail of the file is assumed to be zero. If it is smaller, then any data in the tree beyond the end of the file is ignored.

3.4.2 Non-persistent Data Structures

BPFS maintains two types of data structures for which it does not provide persistence. One, the allocation bitmaps and link counts discussed above, is derived from persistent data structures and used to improve performance. The other tracks ephemeral file system state.

BPFS tracks ephemeral file system state using volatile memory. An example of this is that BPFS keeps a list of freed and allocated blocks from an in-flight copy-on-write when it uses an epoch-based cache (described in Section 3.5.4). Consider a file write. BPFS will remember which blocks were allocated and which need to be freed if the operation succeeds. When the operation completes, BPFS iterates over either the freed or allocated list, depending on the success of the operation, and marks these blocks as free in the volatile block bitmap. Because commits are atomic, this data does not need to be stored in persistent memory or reconstructed. More examples of ephemeral file system state are directory cache and inode cache entries for each directory and inode opened by users. BPFS constructs these cache entries as they are accessed. They mirror the contents stored in BPRAM and track ephemeral state. They mirror, rather than point to, BPRAM data to integrate with the existing file system structures. Use counts and locations in system hash tables are examples of the ephemeral state these structures store.

3.4.3 Software

Our BPFS prototype implementation runs as a Linux FUSE [15] application. To measure run-times of benchmarks we also evaluate a second prototype, implemented in the Windows Driver Model running in Windows Vista; it is similar to our FUSE implementation and is described in our SOSP 2009 paper [9]. Our microarchitectural simulations use a Windows user-level prototype.

All three BPFS implementations are built around a general framework for applying changes to the tree data structures. These functions traverse the tree data structures to perform reads and writes; we call this framework the crawler. The two most-used crawler interfaces allow the caller to execute code on an inode structure or on a range of data blocks in an inode. These callback functions read, modify, or copy-on-write the inode(s) or data leaves. The crawler extends the size of the file as needed for the visit range, informs each callback whether it must copy-on-write (because there are other blocks in the crawl range) or may write in place, and propagates any block number changes. Because BPFS is composed of two levels of tree data structures (i.e., the inode file and all other files), internally the crawler invokes itself to descend the root inode and then the target inode.

We found the crawler useful because it allowed us to incrementally reduce and eliminate copy-on-writes. For example, callers of the crawler do not need to be aware of whether only in-place writes or also copy-on-writes were used. It also allowed us to implement copy-on-write propagation once; we found this useful because this is the most complicated code in BPFS. At the same time, we expect that an implementation that makes its in-place writes without this abstraction could descend trees fewer times, thereby performing fewer memory reads and less computation.

One particular debug feature discovered many of our implementation bugs along the way. At the end of each file system operation, BPFS can check that the volatile allocation records agree with the persistent data structures. (This is the same scan it performs to initially construct the volatile data structures at mount.) Because of the copy-on-write nature of the file system, this check often exposes errors like a dropped update, corrupted metadata structure, or incorrect tree calculation. Additionally, it would frequently inform us of an error at the end of the incorrect operation. (Though, it would only tell us at the end of the incorrect operation; tracking down the precise cause could still require additional thinking and/or legwork.)

3.4.4 Multiprocessor Operation with Epoch Barriers

BPFS is designed to use caches in either write-through or epoch barrier modes. With a write-through cache, each memory write is made durable synchronously. BPFS guarantees that file system operations are committed to BPRAM in global order. With an epoch-based cache, BPFS informs the hardware of its per-CPU ordering constraints. BPFS guarantees that file system operations are committed to BPRAM in program (but not necessarily global) order. On a uniprocessor system in epoch barrier mode, this means BPFS must issue epoch barriers between commits that hardware is not allowed to reorder. In an append operation, it issues a barrier between writing the data and updating the file's size. Another example is that BPFS issues a barrier after each file system operation. On a multiprocessor system we must also consider cases where multiple CPUs contain uncommitted epochs. We describe these issues and our solutions here. We describe the epoch barrier hardware in Section 3.5.4.

On a multiprocessor, BPFS ensures that only one thread modifies any particular address or data structure at a time. Later, another processor may also modify this data. Because the data may still be in cache, the epoch hardware detects this case and serializes the two epochs. To allow CPUs to otherwise operate independently, this is the only ordering guarantee that the epoch barrier hardware provides for epochs on different CPUs. This limited ordering guarantee is the source of the following four issues that BPFS must protect against.

First, because threads can be preempted and rescheduled, the thread executing a single file system operation may end up executing on multiple CPUs. Thus BPFS must ensure that when it is preempted it is rescheduled on the same CPU. This guarantees that all epochs generated within the file system operation are committed in the order in which they were generated.

Second, a thread can switch CPUs between two different file system operations. To provide program-order consistency guarantees, these two file system operations must be committed to BPRAM in the order they were issued. To do so, BPFS tracks the most recent commit write made by each thread. The commit write is always the sole member of its epoch. (Our Windows Driver Model implementation differs slightly on this point. It uses BPRAM non-persistently to track free blocks and inodes and synchronizes using its writes to these data structures.) When a thread executes a file system operation, BPFS reads from the synchronizing BPRAM location. This read creates a dependency that causes the old CPU's data to be flushed, thus guaranteeing that the updates will be committed in program order. Less efficient alternative techniques to provide this ordering include executing file system code on a fixed CPU, monotonically increasing epoch IDs across all CPUs, flushing all cache lines, flushing the relevant cache lines, and issuing enough epoch barriers to flush the in-flight epochs (8, in our evaluated system) with each new epoch containing one otherwise meaningless write.

Third, BPFS caches some data in volatile data structures. When persistent data is cached in volatile memory, two threads that access the volatile cache might not generate accesses to common locations in BPRAM; as a result, the hardware will be unaware of the dependency. BPFS can protect the corresponding persistent structures by writing to their first word to inform the hardware of accesses.

The current implementations of BPFS with epoch barriers do not yet enforce these three constraints. The first two constraints do not affect our write traffic evaluations because they only issue additional reads. Our write traffic evaluations do not account for the writes necessary for the third constraint; for this reason, our evaluations only measure performance for BPFS using write-through caches. Our time-based evaluations are unaffected because these constraints are only relevant during a power failure. In addition, our SOSP 2009 analytical evaluation of BPFS does account for most of their overhead (an 8-byte read for each file system operation) [9]. (It accounts for all overheads of the first two constraints and issues writes to blocks and inodes for the third constraint. It does not issue writes to directory entries.)

Fourth, two threads on different CPUs may update two different locations in the file system. Traditional journaling or shadow paging file systems guarantee that such operations are committed in temporal order by creating a single, total ordering among operations. BPFS with a write-through cache also creates a total ordering of updates. However, BPFS with an epoch-based cache does not; instead, it allows concurrent, in-place updates to different portions of the file system tree. As a consequence, if two threads execute sequentially on two different CPUs, then their updates may be committed to BPRAM in any order with respect to each other. If a total ordering is required, then an explicit synchronization (e.g., `fsync`) must be used to flush dirty data from the CPU's cache. Down the road, as computers gain more CPUs, Featherstitch's patchgroups might become useful to limit the cache flushes to only the required CPUs. As far as we are aware, operating systems have always provided the same ordering guarantees for file system operations made within and across threads. However, we believe this division of work, which distinguishes between intra- and inter-thread file system operations, provides a useful balance of performance and ordering guarantees for BPRAM.

3.4.5 Limitations

One limitation of BPFS compared to a journaling file system is that atomic operations that span a sparse, broad cross-section of the tree can require a significant number of extra copies. The `rename` operation is the sole operation that can still make such updates after applying our optimizations; its writes can span a large portion of the tree in order to update two directory entries. The `write` operation can impose large copy-on-write overheads for small writes. For example, to make a 2 B write that straddles two blocks, BPFS will copy-on-write the remaining 8190 B in the two blocks and, assuming the two data blocks have the same parent indirect block, the parent block's 4096 B. That said, most existing file systems commit data at the granularity of blocks, so they have similar overheads.

Our current prototypes do not yet support memory-mapped files. However, we believe that it would be straightforward to support this feature in the kernel implementation by either copying data into DRAM and occasionally flushing it out to BPRAM (as is done by current disk-based file systems), or by mapping BPRAM pages directly into an application's address space. In the latter case, atomicity and ordering guarantees would not be provided when writing to the memory-mapped file, but the file's data could be accessed without a trap to the kernel. Wear leveling will be required to ensure that malicious programs cannot burn out the BPRAM device; we will discuss these issues further in the next section.

Another limitation is the overall interface BPFS provides to BPRAM. Rather than implementing a new file system, we could offer the programmer a fully persistent heap. However, this approach has the disadvantage of requiring significant changes to applications, whereas BPFS allows existing programs to reap the benefits of BPRAM immediately. In addition, the file system interface provides a well-known abstraction for separating persistent data from non-persistent data, and it allows the file system to enforce consistency in a straightforward manner. Our current design provides a balance between performance, consistency, durability, and backward compatibility, but we also believe that persistence within the user-level heap will be a fruitful area of future research.

3.5 Hardware Support

In this section we discuss the atomic 8-byte write primitive and the epoch barrier caching mode that we add to hardware. First, we discuss the details of phase change memory, which we believe is currently the most promising form of BPRAM. Second, we discuss wear leveling and write failures for phase change memory. Third, we show how we enforce atomicity with respect to failures. Finally, we show how one can modify the cache controller and the memory controller to enable buffering and reordering while still enforcing file system ordering constraints.

3.5.1 Phase Change Memory

Phase change memory, or PCM, is a new memory technology that is both non-volatile and byte-addressable; in addition, it provides these features at speeds within an order of magnitude of DRAM [12, 33]. PCM encodes data into resistivity (DRAM uses electrical charge). Its chalcogenide glass can be heated to 650°C then and cooled, either slowly or rapidly, to switch the glass between crystalline and amorphous phases, encoding a 0 and 1, respectively.

PCM cells can be organized into an array structure much like that of DRAM [3]. Thus, it is possible to manufacture a PCM DIMM that operates in much the same way as an existing DRAM DIMM, albeit with different timing parameters and access scheduling constraints [33]. At a minimum, memory controllers could support PCM DIMMs by modifying the timing parameters of an existing DDR interface. We propose two additional modifications to help us construct robust and performant software on top of non-volatile system memory.

For this dissertation, we assume that the PCM-based storage system is organized as a set of PCM chips placed in DDR-compatible DIMMs. One limitation of this approach is that capacity will be restricted by the density of the chips residing on a DIMM. For example, a 2008 Samsung prototype PCM chip holds 512 Mb [33], so with 16 chips on a high-capacity DIMM, a capacity of 1 GiB per DIMM is feasible. Combined with process technology and efficiencies from manufacturing at volume, which will further improve density and capacity, we expect to have enough capacity to provide a useful storage medium in the near future. If additional capacity is required, we can place larger quantities of PCM (hundreds of gigabytes) on the PCI Express bus in addition to the PCM on the memory bus.

We also assume that PCM provides bandwidths and latencies close enough to those of DRAM that eliminating the DRAM-based buffer cache does not harm performance. If, instead, PCM is slower than DRAM, we should revisit this choice. For example, slower memory reads could harm the execution performance of memory-mapped binaries. Slower writes could stall system calls that write to the file system, especially for short-lived files, relative to file systems with a DRAM buffer cache. It may make sense to issue writes asynchronously using, for example, epoch barriers or to queue them into a DRAM-based buffer.

3.5.2 Wear Leveling and Write Failures

Although PCM has much higher write endurance than NAND flash, it will still wear out after a large number of writes to a single cell. The industry consensus as of 2007 was that PCM cells will be capable of enduring at least 10^8 writes in 2009 and up to 10^{12} by 2012 [12]. Even though these endurance figures are high compared to other non-volatile memories, placing PCM on the memory bus instead of an I/O bus may expose the cells to greater write activity and thus require *wear leveling*, which is a process that distributes writes evenly across the device to reduce wear on any single location. Although our file system does not specifically concentrate updates on one location (recall that most updates are committed locally, not at the file system root), there is the potential for some workloads to result in "hot" locations.

Fortunately, there are several approaches to wear leveling that can operate independently of our file system. First, we can design PCM arrays in ways that minimize writes, extending device lifetime from 525 hours to 49,000 hours (5.6 years) [33]. Second, several mechanisms have been proposed for applying wear leveling to PCM [56, 92]. In short, effective wear leveling can be implemented by using two techniques: within each page, wear is evened out by rotating bits at the level of the memory controller, and between pages, wear is evened out by periodically swapping virtual-to-physical page mappings. By choosing these shifts and swaps randomly, additional defense against malicious code can be provided. These show that it is possible to design reasonable wear-leveling techniques that are independent of BPFS.

When eventual failures occur, we expect to detect them using error-correcting codes implemented in hardware. For example, we can take advantage of existing error-correcting codes used for flash [43]. When PCM pages degrade beyond the ability to correct errors in hardware, the operating system can retire PCM pages, copying the data to a new physical page and then updating the page table. Of course, data can still be lost if sufficiently many bits fail; however, in this dissertation, we assume that PCM hardware will be designed with enough redundancy to make such failures negligibly rare.

3.5.3 Enforcing Atomicity

To guarantee atomicity for 8-byte writes, the hardware must ensure that, in the case of a power failure, a write either completes entirely, with all bits updated appropriately, or fails entirely, with all bits in their original state.

We propose enforcing atomicity by augmenting DIMMs with a capacitor holding enough energy to complete the maximum number of write transactions ongoing within the PCM subsystem. Since all writes are stored temporarily in volatile row buffers on each DIMM before being written to PCM, having a capacitor on each DIMM ensures that all writes residing in the row buffers are completed. After losing power the memory controller will fail to issue further commands, but this will ensure that any in-progress writes will complete. No 64-bit word will be left in an intermediate state.

Note that unrecoverable bit failures can occur while performing the final writes during a power failure. As above, we assume that PCM devices provide enough redundancy to make such failures extremely unlikely. If additional reliability is required, the memory controller can be modified to write all in-flight writes to a backup location as well as to the primary location in the event of a power failure. This approach increases the chances of successful completion at the expense of additional capacitance.

The amount of power required to complete all in-flight writes is quite small, even for a mobile device. To write a logical zero, a PCM bit requires a current ramp down from 150 μA to 0 μA over 150 ns, requiring 93.5 nF at 1.2 V. Similarly, to write a logical one, a PCM bit requires 300 μA over 40 ns, requiring 75 nF at 1.6 V. Assuming PCM row widths of 512 bits (one cache line), the total capacitance required would vary between 38,400 and 47,800 nF. To maintain stable power, the capacitor would need to be somewhat larger, with circuitry to provide a transient but stable output voltage as the capacitor discharges. On-chip decoupling capacitors can provide part of this charge; the total decoupling capacitance on the Alpha 21264 was 320 nF and the Pentium II contained 180 nF [52]. Discrete capacitive elements on the memory module can easily provide several thousand nF of supplemental charge [65].

If desired, larger units of atomicity could be provided by integrating additional capacitors at the board level. We propose 64 bits because a single atomic pointer update can be used as a primitive in order to update even larger quantities of data, as shown in BPFS.

3.5.4 Enforcing Ordering

Cache coherence protocols and memory barriers (e.g., the x86 `mfence` instruction) suffice to ensure that all CPUs have a consistent global view of DRAM. As long as hardware provides this consistent view, it does not matter when or in what order data is actually written back to DRAM. For example, if writes *A* and *B* are separated by an `mfence`, the `mfence` only guarantees that *A* will be written to the cache and made visible to all other CPUs via cache coherence before *B* is written to the cache; it does not ensure that write *A* will be written back to DRAM before write *B*. When using memory writes to modify persistent data, though, the order in which writes are made to DIMMs is now important. For example, consider the sequence of operations to overwrite a 4 kiB file stored in BPFS. First, a new 4 kiB block is allocated in BPRAM and the updated data is written to that block. Then, a block pointer in the file system tree is updated from the previous to the new block. With a write-back cache, these bytes are likely to be dirty in the L1 or L2 cache; they have not yet been written back to BPRAM. If the cache controller chooses to write back the block pointer update before it writes back the 4 kiB block, the file system in BPRAM will be inconsistent; the file system tree will point to unintended and uninitialized data. This inconsistency will not be visible to any currently-executing code, since existing cache coherence and memory barrier mechanisms ensure that all CPUs see the updates in the correct order. However, if a power failure occurs before all bytes are written back to BPRAM, the file system will be inconsistent when the machine is rebooted. Thus, in order to ensure that the file system in persistent memory is always consistent, we must respect any ordering constraints when data is written to persistent memory.

The simplest approach to enforce these constraints is to disable write buffering for BPRAM addresses; in this mode, the caches and memory controller preserve the order of memory writes. In particular, on x86 one can set the address range to write-through mode using MTRR or PAT entries [25, Chapter 11]. Current systems use this mode for memory-mapped I/O (MMIO), which uses memory reads and writes to exchange messages with external hardware. We have designed BPFS to use two cache modes; this is the first. In this mode BPFS guarantees that file system operations commit synchronously; that is: they are made durable in file system operation order and they become durable before the operation returns. This is simple for software to use, and is available in current hardware.

The drawback of write-through caching is that the caches and memory controller can provide higher throughput and lower latency when they may buffer and reorder writes. A second possibility is for BPFS to flush the entire cache only at memory barriers; this would ensure that all data is given to non-volatile memory in a correct order. However, flushing the cache is also quite costly in terms of performance, and would have the side-effect of evicting the volatile working sets of any other applications sharing the cache.

A third possibility is for software to flush only the necessary cache lines, rather than all volatile and persistent writes. To do this, BPFS would need to track each persistent cache line it modifies, flushing these before each write that commits. On x86, these flushes can be accomplished by issuing an `mfence` instruction and then an appropriate set of `clflush` instructions. However, we expect that tracking the set of dirtied cache lines is difficult both in terms of software complexity and system performance. We also feel that this represents a poor division of labor between software and hardware, that software must do a large amount of work to compensate for the deficiencies of what we think should, ideally, be a transparent caching mechanism.

Epoch Hardware Interface

Therefore, we propose a fourth alternative: allow software to explicitly communicate ordering constraints to hardware. This allows hardware to cache writes to persistent data without also forcing software to micromanage when caches flush data. Additionally, this approach allows hardware to cache data for longer periods and allows CPUs to operate with greater independence than software cache flushes permit.

We call this mechanism an *epoch barrier*. An epoch is a sequence of writes to persistent memory from a single CPU, delimited by this new form of memory barrier issued by software. An epoch that contains data that is not yet reflected to BPRAM is an *in-flight* epoch; an in-flight epoch *commits* when all of its associated dirty data is successfully written back to persistent storage. The key invariant is that when a write is issued to persistent storage, all writes from all previous epochs must have already been committed to the persistent storage, including any data cached in volatile buffers on the memory chips themselves. So long as this invariant is maintained, an epoch can remain in-flight within the cache subsystem long after the processor commits the memory barrier that marks the end of that epoch, and multiple epochs can potentially be in flight within the cache subsystem at each point in time. Writes can still be reordered within an epoch, subject to standard reordering constraints.

When using epoch-based caching, BPFS issues an epoch barrier immediately before a write that commits and which depends on other writes issued after the previous epoch barrier. It also issues an epoch barrier immediately after the final write that commits a file system operation, so that the writes from later operations follow it.

The epoch barrier mechanism is similar to the mechanisms disks provide for software to specify ordering constraints among in-flight writes. Disks use these ordering mechanisms to address analogous caching limitations with the modes of on-disk caches. (SCSI's TCQ task attribute is an example of this kind of ordering mechanism [83, Chapter 8].) We believe that this similarity with vetted and widely-used ordering mechanisms speaks well for the design of epoch barriers. Their most significant difference is that SCSI TCQ is specific to a target (roughly, a disk drive), while epoch barriers are specific to a CPU. We designed epoch barriers to be CPU-specific because of PCM's greater performance relative to CPUs; we think this choice is important in enabling CPUs to independently interact with PCM storage. Epoch barriers are independent of, say, the target DIMM because we expect systems to typically use one BPRAM file system at a time.

Existing storage systems also offer an additional mechanism that software can use: the disk sends a response after each write becomes durable and software only submits a write request when it is safe for it to commit (§2.5 elaborates). The reasons we chose the former of these two approaches include that the latter requires software manage its own write-back cache; we expect these duplicated writes and the cache maintenance would be performance bottlenecks. Additionally, the latter approach builds on the request and request response architecture that current storage interfaces use; in contrast, memory accesses are made directly.

Epoch Hardware Modifications

Our proposed hardware support includes minor modifications to several parts of the PC architecture. We discuss how our modifications impact the processor core, the cache, the memory controller, and the non-volatile memory chips.

First, each processor must track the current epoch to maintain ordering among writes. Each processor is extended with an *epoch ID counter* for each hardware context, which is incremented by one each time the processor commits an epoch barrier in that context. Whenever a write is made to an address located in persistent memory, it is tagged with the value of the current epoch ID, and this information is propagated with the write request throughout the memory system. The epoch ID counter is 64 bits wide, and in a multiprocessor system, the epoch ID space is partitioned among the available hardware contexts (effectively using the top bits as a context ID). Thus, epoch IDs in a shared cache will never conflict.

Next, each cache block is extended with a *persistence bit* and an *epoch ID pointer*. The persistence bit indicates whether or not the cached data references non-volatile memory, and it is set appropriately at the time a cache line is filled, based on the address of the block. The epoch ID pointer indicates the epoch to which this cache line belongs; it points to one of eight hardware tables, which store bookkeeping information for the epochs that are currently in-flight.

We then extend the cache replacement logic so that it respects the ordering constraints indicated by these epoch IDs. The cache controller tracks the oldest in-flight epoch resident in the cache for each hardware context, and it considers any cache lines with data from newer epochs to be ineligible for eviction. In cases where a cache line from a newer epoch *must* be evicted, either because of a direct request or because no other eviction options remain, the cache controller can walk the cache to find older cache entries, evicting them in epoch order. The cache maintains bookkeeping information for each 4 kiB block of cache data in order to make it easy to locate cache lines associated with each in-flight epoch.

This cache replacement logic also handles two important corner cases which indicate that epochs on different CPUs must be ordered. First, when a processor writes to a single cache line that contains dirty data from a prior epoch, the old epoch must be flushed in its entirety—including any other cache lines that belong to that older epoch. Second, when a processor reads or writes a cache line that has been tagged by a different hardware context, the old cache data must be flushed immediately. This requirement is particularly important during reads in order to ensure that we capture any read-write ordering dependencies between CPUs.

Note that the coherence protocol does not change; existing coherence protocols will work correctly as long as our cache replacement policy is followed.

Finally, the memory controller must also ensure that a write cannot be reflected to PCM before in-flight writes associated with all of the earlier epochs are performed. To enforce this rule, the memory controller records the epoch ID associated with each persistent write in its transaction queue, and it maintains a count of the in-flight writes from each epoch. When each write completes, it decrements this counter, and it does not schedule any writes from the next epoch until the current epoch's counter hits zero.

The overall modifications to the hardware include four changes. First, we add one 64-bit epoch ID counter per hardware context. Second, we extend the cache tags by 4 bits: 3 for the epoch ID pointer and 1 for the persistence bit. Third, we augment the cache with 8 bit-vectors and counter arrays for fast lookup of cache lines in a given epoch (total overhead of 7 kiB for a 4 MiB L2 cache). Finally, we add capacitors to ensure that in-progress writes complete. The total area overhead for a dual-core system with 32 kiB private L1 caches, a shared 4 MiB L2 cache, and a maximum of 8 in-flight epochs is approximately 40 kiB.

We believe that these changes are neither invasive nor prohibitive. Although the capacitor requires board-level support, it applies well-known power supply techniques to ensure a temporary but stable supply voltage in the event of a power loss. Additionally, the changes do not harm performance on the critical path for any cache operations. For example, the cache changes affect the replacement logic in the control unit only; they do not affect the macro blocks for the cache.

3.6 Evaluation

In this section we evaluate the performance of BPFS relative to other file systems, the effectiveness of the BPFS performance optimizations, the correctness of these optimizations, and the effectiveness of epoch barriers.

We compare BPFS with other file systems which were originally designed for disks. These *comparison file systems* are ext2, ext3 and ext4 in ordered and data journaling modes, and Btrfs.

BPFS's per-system-call durability and consistency guarantees will have a performance cost, relative to a file system with weaker guarantees. Our goal is for BPFS on PCM to still run faster than the comparison file systems *on disk*, giving users some performance incentive to switch to BPFS (as well as consistency incentives).

However, without PCM we cannot measure performance in units of time. Therefore, instead we measure the number of bytes a file system writes, since this value is a major factor of performance for a file system backed by PCM. Conservatively, write throughput to PCM is expected to be at least 3 times greater than the maximum sustained sequential write throughput to disk. Condit et al. [9] report a 322 MiB/s sustained throughput for the PostMark benchmark with BPFS running in simulation on PCM accessed via a DDR2-800 memory bus and a write-through cache. (We evaluate with a write-through cache instead of the faster epoch barriers because a write-through cache provides stronger durability and consistency guarantees.) VanDeBogart et al. [79] observe the maximum sustained throughput of a current disk as 108 MiB/s. Thus, if BPFS writes 3 or fewer times the number of bytes compared to another file system, we expect running BPFS on PCM will perform no more slowly than that file system does on disk.[2]

Our performance hypotheses are therefore:

1. BPFS provides its improved guarantees while also writing no more than 3 times as many bytes as the comparison file systems.

2. Each BPFS optimization significantly reduces the number of bytes that at least some file system operation writes.

We also inspect several benchmarks and verify that, at least for these benchmarks, the BPFS optimizations maintain commit atomicity. Finally, we evaluate the effectiveness of epoch barriers vs. write-through caching.

3.6.1 Experimental Setup

The experimental setup consists of BPFS implemented as a FUSE [15] process running on 64-bit Ubuntu 10.04 (Linux kernel 2.6.32). BPFS maps a preallocated file into its address space as its persistent store. We measure the number of bytes that BPFS writes during a given run as the number of bytes that CPU instructions write to the mapped memory region.[3] To do this we run BPFS in the dynamic binary instrumentation tool Pin [37], revision 36111, with a Pintool we developed for this purpose.

We measure the performance of the comparison file systems on a Dell Precision 380 computer with 1 GiB of RAM, a hyper-threaded 3.20 GHz Pentium 4 CPU, and a 7200 RPM, 500 GB, SATA2 Seagate ST3500320AS test disk. We measure the number of bytes these file systems write to their persistent store as the number of disk sectors written to the test partition. To do this we read `/proc/diskstats` immediately after mounting the file system and then after the experiment completes and the file system is synced, and multiply the number of disk sectors by the number of bytes that make up a sector.

The strength of the guarantees that the Linux file systems and BPFS provide is ordered as "ext2 $<$ ext3-ordered $=$ ext4-ordered $<$ Btrfs $=$ ext3-data $=$ ext4-data $<$ BPFS." Figure 3.5 compares the file systems in detail. We include ext3 and ext4 because they are the two most widely used Linux file systems. We evaluate both in ordered journaling and data journaling modes. Ordered journaling mode is the default mode for most Linux distributions. The Btrfs file system is an in-development shadow paging file system [5]. Finally, we include ext2 to compare BPFS with a file system that provides no guarantees. We use the default

[2]If PCM is even faster, then we also expect BPFS performance on PCM to fare even more favorably.

[3]An alternate measurement would group writes into cache lines and measure the number of cache line writes across the memory bus instead. Since this measurement depends on the cache mode, CPU store buffers, and other implementation constraints, we do not consider this alternative further.

File system	**Durable**	**Atomic**			**Ordered**	
		Metadata	Data (block)	Data (syscall)	Metadata	Data (syscall)
ext2	–	–	–	–	–	–
ext3 – ordered	–	✓	–	–	✓	–
ext4 – ordered	–	✓	–	–	✓	–
Btrfs	–	✓	✓	–	✓	✓
ext3 – data	–	✓	✓	–	✓	✓
ext4 – data	–	✓	✓	–	✓	✓
BPFS	✓	✓	✓	✓	✓	✓

Figure 3.5: Commit guarantees for a file system operation. A **durable** operation commits before it returns, an **atomic** operation commits completely or not at all, and an **ordered** operation commits in invoked order. Data operations can be atomic with respect to the entire operation or only individual blocks. More check marks means stronger guarantees.

writeback trigger cache sizes and intervals for all file systems. These triggers can vary across file systems; for example, ext3 and ext4 default to a commit interval of 5 seconds while Btrfs defaults to 30 seconds. For these tests, increasing the commit interval of ordered ext3 and ext4 to match the longer interval used by Btrfs results in ext3 and ext4 writing 95–100% the number of bytes they write with their default interval.

All file system images are 6 GiB.

3.6.2 Writes for BPFS and Comparison File Systems

This section tests our first performance hypothesis. It shows that BPFS writes fewer than 3 times the number of bytes that at least some comparison file system writes in each macrobenchmark and that BPFS writes fewer than this factor for all comparison file systems in most macrobenchmarks. Results are the mean of 5 runs and are shown with min/max error bars in Figure 3.6.

The *untar benchmark* extracts version 2.6.15 of the Linux kernel from a tar archive. This creates a large number of files and appends 203 MiB of file data to these files. The *delete benchmark* deletes the extracted Linux kernel. The *compile benchmark* extracts version 2.0.63 of the Apache webserver [1], runs its configure program, and runs make to build the software. The ld and gold variants use these two GNU binutils linkers. The *PostMark benchmark*, version 1.5, emulates small file workloads seen on email and netnews servers [29]. We evaluate two PostMark configurations. Both configurations use 4 kiB blocks for reads and writes. The small configuration creates files with sizes in the range of 512 B to 1 MiB and initially creates 100 files. We use the default PostMark values for the remaining settings. This configuration writes about 222 MiB and reads about 161 MiB to and from the file system. The large configuration creates files with sizes in the range of 512 B to 16 MiB and initially creates 500 files. As with the small configuration, we use the

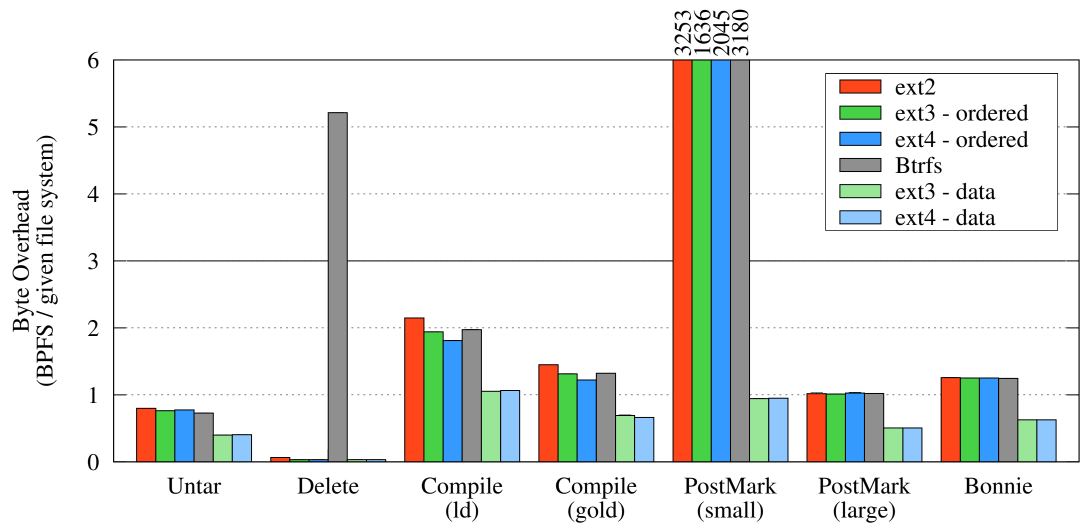


Figure 3.6: Number of bytes that BPFS writes, relative to comparison file systems, for several macrobenchmarks. Y-axis values greater than 1 indicate that BPFS writes more bytes and values less than 1 indicate that BPFS writes fewer than the given file system. Shorter bars are better for BPFS.

default PostMark values for the remaining settings. This configuration writes about 6.7 GiB and reads about 2.1 GiB to and from the file system. Finally, the *Bonnie benchmark* creates, reads, and deletes files. We use Bonnie++ version 1.96 [4].

These macrobenchmarks are meant to represent or mimic common file system workloads. Each is commonly used to evaluate file system performance. We do not include macrobenchmarks that vary their workload in testing performance per unit of time, since we measure byte overhead. Additionally, we sought workloads that stress the performance bottlenecks of BPFS relative to existing file systems.

BPFS writes more than 3 times the number of bytes that other file systems write for two macrobenchmarks. In both cases, this is because BPFS commits synchronously while existing file systems batch commits. We first describe the small PostMark results; BPFS has the largest write factor relative to the other file systems for the this macrobenchmark. Although PostMark writes about 222 MiB of file data to the file system, the ext2, ordered ext3 and ext4, and Btrfs file systems write only 72–132 kiB to disk; these file systems write fewer bytes to disk than PostMark writes to them. They do so because they batch commits. With batching, these file systems do not commit blocks that were modified after the previous commit but then unreferenced before the next commit. Such file data writes become no-ops. This optimization applies to the small PostMark macrobenchmark because it writes no more data than fits in the operating system's buffer, makes no durability requests (e.g., no `fsync` calls), deletes all of its files, and runs for only a short period (about 1.5 seconds). BPFS cannot avoid such persistent writes because it commits each file system operation synchronously. BPFS writes about 224 MiB to its persistent store and copy-on-writes zero blocks.

When we alter the small PostMark macrobenchmark to break any one of the requirements for the no-op optimization, BPFS and the file systems with this optimization write a similar number of bytes. Figure 3.6 shows one example with the large PostMark macrobenchmark. Here, PostMark writes more data than fits in the buffer cache (6.7 GiB of data and 1 GiB of DRAM). Requesting durability after each PostMark step, not removing any of PostMark's files, or sleeping for 1 second after each PostMark step results in similar byte-count relationships.

In contrast to the ordered ext3 and ext4 results, in data journaling mode these file systems write about 5% more bytes than BPFS writes for the small PostMark macrobenchmark. Although the file systems continue to employ the no-op optimization, in this mode they also write all file data to their journal data structures. These are managed by the Linux journaling modules jbd and jbd2, which cannot delete journal entries. These journaling modules can only mark entries as stale so that they do not copy the blocks into the live file system. Therefore, data journaling ext3 and ext4 write all modified blocks at least once to persistent storage. This results in these file systems writing a similar number of bytes as BPFS writes for small PostMark. For large Postmark, they write about twice as many bytes as BPFS writes. As with ordered journaling mode, here PostMark writes more data than fits in the buffer cache. This forces intermediate commits. Most modified blocks are thus committed before they are unreferenced, and the file system writes the file data blocks to both the log and the live file system.

While BPFS writes more than 3 times the number of bytes that most of the tested file systems write for the small PostMark macrobenchmark, two notes may affect the interpretation of these results. The PostMark benchmark is designed to emulate email and netnews workloads; these workloads typically include durability requirements. PostMark, however, does not. The durability requirements of such workloads forces file systems to issue writes for the file data blocks that the no-op optimization can omit. However, other workloads do work with ephemeral files. For these we expect that BPFS will write significantly more bytes to persistent storage than existing file systems write. That said, our metric may be overly narrow for such workloads. While we use the number of bytes written as a proxy for run-time, this does not account for time to write to a DRAM-based buffer cache. Although BPFS does not use a buffer cache, the comparison file systems do. These file systems write all file data to the buffer cache, and may later also copy the data to their persistent store. For this reason we expect that such workloads will see BPFS and existing file systems write similar numbers of bytes of file data to PCM and DRAM, respectively. For these cases, if BPFS sees performance with a PCM persistent store that is similar to the performance that existing file systems see with a DRAM buffer cache, the workloads may see similar run-times, even though BPFS provides stronger durability guarantees. If PCM performance is significantly lower than DRAM performance, BPFS could perhaps buffer writes in a DRAM buffer cache and weaken the durability guarantees it provides to applications to improve performance. Another option if PCM is slower than DRAM is for workloads that do not require durability to use file systems that do not provide durability, such as the Linux tmpfs file system.

BPFS writes 5 times the number of bytes that Btrfs writes for the delete macrobenchmark. To delete the 203 MiB of files created by tar, BPFS writes about 334 kiB and Btrfs writes 64 kiB. BPFS copy-on-writes zero blocks. Although the relative difference between these file systems is 5×, we believe the absolute difference is small for both relative to the size of the data set being deleted. We suspect that Btrfs writes fewer bytes because it batches the deletes, allowing it to transform these into a single operation. In contrast, BPFS deletes each file synchronously. For each directory entry BPFS clears the inode field and decrements the inode's link count. At the same time, BPFS writes far fewer bytes than the other file systems. The ext2 file system writes the least of the remaining file systems, at 5 MiB. BPFS writes about 6% of this number of bytes. ext2 writes so many more bytes than BPFS because it one, modifies an additional half dozen data structures and two, writes at the granularity of file system blocks (compared with BPFS writing individual data structure fields). ext3 and ext4 in ordered and data journaling modes work similarly as ext2, but write about twice as many bytes because they also journal metadata writes.

BPFS writes fewer than 3 times the number of bytes for all file systems for the remaining macrobenchmarks. Of these, BPFS's write factor is highest for the compile macrobenchmark with the ld linker. Here, BPFS writes about 2 times the number of bytes that ext2, ordered ext3 and ext4, and Btrfs write. Half of the bytes that BPFS writes are to copy-on-write data blocks. These copy-on-writes are caused by many, small file overwrites. This is the third type of overhead that BPFS exhibits for this set of macrobenchmarks. 52% of the file overwrites request to write fewer than 64 B and 75% fewer than 512 B. Each overwrite requires a copy-on-write for the block (4 kiB), minus the size of the requested write. The sources of these small overwrites are the assembler (4,000 overwrites) and the linker (17,000 overwrites). We also report results with the newer GNU binutils linker named gold, a replacement designed to significantly improve link times for programs with many symbols. In all, gold requests to write about as many bytes as ld, but it makes significantly fewer overwrites: 2,000 instead of 17,000. This reduces the number of bytes that BPFS writes from 126 MiB to 85 MiB (copy-on-writes from 59 MiB to 13 MiB). Switching from ld to gold reduces the write factor for BPFS from 2 to 1.5 times that of ordered ext3 and ext4. Further, with gold BPFS writes fewer bytes than Btrfs and the ext3 and ext4 file systems in data journaling mode. BPFS writes about the same number of bytes as other file systems write for the Bonnie macrobenchmark (5.0 GiB). It copy-on-writes 1.0 GiB. BPFS writes slightly fewer bytes than other file systems write for the untar macrobenchmark. It copy-on-writes zero blocks.

3.6.3 DRAM Comparisons

The previous section used the number of bytes a file system writes as the performance metric to compare the performance of BPFS with existing file systems. Although we believe that writes to PCM will be the largest factor in file system run-time, other factors also play a role. These include code execution, DRAM accesses, and persistent store read accesses and latency. In this section we measure run-times with DRAM as a substitute for PCM to compare the file systems with all overheads except those due to PCM. We implement BPFS as a Windows kernel file system and back it with DRAM. For our benchmarks BPFS-Kernel is faster than NTFS when NTFS is backed by either a hard disk drive or DRAM. Therefore we do not believe that factors other than PCM access times and latency will limit the performance of BPFS on PCM relative to the comparison file systems.

Experiment Setup The experimental setup for our run-time evaluation differs from the setup described in Section 3.6.1. We run these experiments on a system with two dual-core 2 GHz AMD Opteron CPUs with 32 GiB of RAM and running 64-bit Windows Vista SP1. We evaluate a second implementation of BPFS as a Windows kernel file system; we call this BPFS-Kernel. BPFS-Kernel runs by allocating a contiguous portion of RAM as its "PCM" to store all file system data structures, as the FUSE-based BPFS does. All other data structures that would normally reside in DRAM (e.g., the directory cache) are stored through regularly allocated memory within the kernel. NTFS backed by DRAM uses a RAM disk; this system is meant to represent an alternative file system where we simply run an existing disk-based file system in persistent memory. We use the RAM disk driver RAMDisk, version 5.3.1.10 [55] and call this system NTFS-RAM. Both BPFS and NTFS-RAM run with CPU caches in write-back mode. NTFS backed by disk uses two 250 GB, 7200 RPM, 8 MiB cache Seagate Barracuda disks connected via an NVIDIA nForce RAID controller configured in RAID-0 mode. We call this system NTFS-Disk. We measure time using `timeit`, which captures the wall clock time for each benchmark. Unless otherwise noted, results are the mean of 5 runs, and error bars represent 90% confidence intervals.

PostMark First we evaluate a benchmark similar in spirit to the PostMark macrobenchmark. License issues prevented us from using the original PostMark source code that we used elsewhere in this evaluation, so we wrote our own version of the benchmark. Our PostMark-like macrobenchmark creates 100 files, executes 5000 transactions on those files consisting of reads and writes, and then deletes all files. This benchmark serves as a test of file system throughput, since it does not involve additional computation.

The results are presented in Figure 3.7a. The first bar shows the time to execute the benchmark on NTFS-Disk, while the third bar shows the time to execute on BPFS-Kernel. NTFS backed by disk takes 3.2 times the run-time of BPFS-Kernel. One reason for this result is that when a file is created, NTFS does a synchronous write to disk to commit a journal operation, whereas NTFS-RAM and BPFS-Kernel have no such overhead. The second bar shows the performance of NTFS backed by a RAM disk. NTFS-RAM takes 1.7 times the run-time of BPFS-Kernel. We suspect that a considerable portion of this overhead is due to NTFS writing all file data twice, once to the DRAM buffer cache and once to the DRAM persistent store, and writing metadata a further additional time, to the journal in the DRAM persistent store.

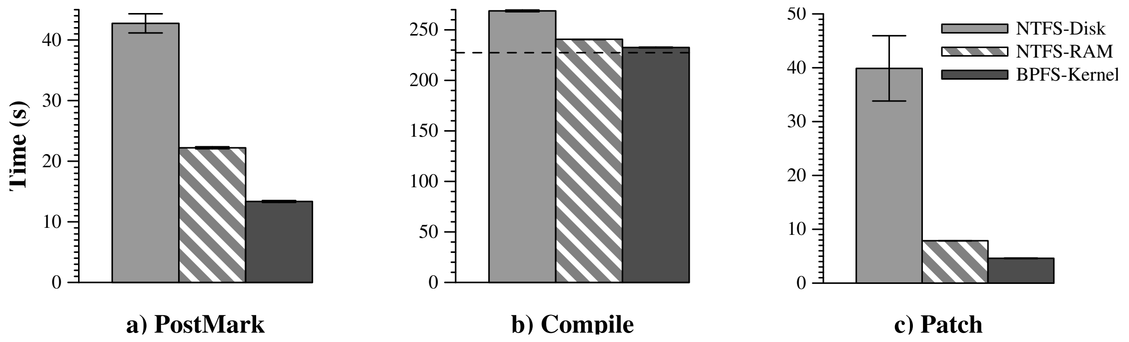


Figure 3.7: Run-times for BPFS-Kernel and NTFS macrobenchmarks. The dashed line for compile indicates the approximate amount of time in computation. Lower bars are better.

Compile We repeat the compile benchmark to see how BPFS-Kernel compares to NTFS on a benchmark that overlaps computation with file system I/O. We compile Apache version 2.0.63, as in Section 3.6.2. Note, however, that the build target and build tools differ.

Figure 3.7b shows the results. BPFS-Kernel executes the benchmark 13% more quickly than NTFS backed by a disk, and 3.4% more quickly than NTFS backed by a RAM disk. This improvement in performance is much lower than with the other run-time benchmarks. We believe this is the case largely because only 5.1 seconds of time was spent executing file system operations; the remainder of the time was compute bound. Figure 3.7b shows a dashed line to indicate the best-case performance of a file system on this benchmark. Even if file operations were instantaneous, the maximum speedup over NTFS backed by a RAM disk is 6.5%.

Patch Our last benchmark for BPFS-Kernel on DRAM decompresses the Apache source tree and then runs a patch operation against the tree. The patch script examines each file in the source tree, looks for a certain pattern, replaces it as necessary, and then writes the entire file out to the file system, replacing the old version. This benchmark strikes a balance between the throughput-bound PostMark-like benchmark and compute-bound Compile benchmark.

Figure 3.7c shows the results. NTFS-Disk takes 8.7 times the run-time of BPFS-Kernel and NTFS-RAM takes 1.7 times the run-time of BPFS-Kernel.

3.6.4 BPFS Optimization Effectiveness

In this section we test our hypothesis that each BPFS optimization significantly reduces the number of bytes that at least some file system operation writes. We first evaluate the effectiveness of these optimizations for a range of individual file system operations, then the optimizations for macrobenchmarks to show that the improvements carry over to example workloads.

Microbenchmarks

In this section we evaluate the contributions of the BPFS optimizations to individual file system operations. We measure the number of bytes written for a number of file system operations, operation arguments, and file system states. Results are shown in Figure 3.8. They show that each optimization group can reduce the number of bytes that BPFS writes by more than an order of magnitude; each optimization column contains red cells (90–100% reduction from the previous column). In the following we describe why the results additionally show that each component of the groups reduces overheads by more than an order of magnitude for all but one optimization (parent directory entry), and by at least 50% for all. We discuss the results in order of optimization grouping.

The base point for comparison runs BPFS in shadow paging mode (SP). That is, to overwrite allocated blocks, BPFS copy-on-writes the desired blocks and propagates their updated block numbers to the root of the file system.

Although we test BPFS with only shadow paging commits as our base point, this mode still uses BPFS's layout optimizations. Specifically, the base file system does not contain persistent block and inode bitmaps, places the user ID and group ID inode fields within and aligned to 8 B, and does not contain parent directory entries. We note the inclusion of these layout optimizations because the amount of benefit that the tested optimizations provide depends in part on the existence of these optimizations.

We also note the microbenchmarks do not directly test our performance hypothesis for these three optimizations. Instead, we explain their significance here. Maintaining the block and inode bitmaps would force broader copy-on-writes for operations that allocate or free these objects. For example, with all optimizations, the unlink file microbenchmark makes no copy-on-writes (it writes 16 B). Including a block bitmap would force a copy-on-write of the directory entry, the blocks containing the block bitmap entries for the unlinked file, the block containing the inode bitmap entry for the unlinked file, and some number of indirect blocks to atomically commit the changes. Thus, it would copy-on-write at least four blocks. Second, using unaligned or non-contiguous user ID and group ID inode fields would force the chown microbenchmark to copy-on-write the target inode. This would cause the file system to copy-on-write one block. Third, maintaining the parent directory entries would force broader copy-on-writes for renames that change the parent directory of a directory file. For example, with all optimizations, the rename directory (inter-directory) microbenchmark copy-on-writes at least three blocks. (These are the blocks that contain the source and destination directory entries plus some number of parent directory blocks and some number of inode file blocks.) Including parent directory entries would force the rename to also modify the child's parent directory entry and propagate this change to a three-way common inode file block. Excluding this optimization would approximately double the number of bytes written in many cases.

Figure 3.8 shows that each of the last three optimization groups increases the number of bytes the file system writes for some microbenchmarks by a few bytes (up to 256 bytes or up to 0.6%). There are two causes of these increases. One is allowing in-place writes (a goal of the optimizations) for blocks that are subsequently copied-on-write. Additionally, the middle three columns (+ SC, + A or T, and + A and T) employ a copy-on-write optimization to not write bytes that happen to already have the desired value. This optimization is actually an artifact of our prototype; a real file system might find this write-optimization increases run-time because it requires additional BPRAM reads. The last column (+ Other) does not include this optimization.

Short-circuit commit The second column (+ SC) in Figure 3.8 shows the results of enabling short-circuit commits (§3.3.1). Short-circuiting has its most significant effect on the read and readdir system calls, where it reduces the number of bytes that BPFS writes by more than an order of magnitude. These calls each need to set the new access time for the file. Without short-circuiting, each call must copy-on-write the block that contains the inode and all of the parent blocks for this block. With the atomic 8-byte write that short-circuiting allows, each call directly writes the new access time. More broadly, short-circuiting reduces the number of bytes that BPFS writes by more than 67% for all categories of system calls. While short-circuiting cannot generally avoid all copy-on-writes, it does always reduce the propagation distance of block pointer updates. Additionally, all tested system calls see some benefit from this optimization. The degree of benefit a given system call sees from this optimization is a function of both the number of bytes the system call must modify (e.g., a `write` of a new 128 B sequence must write at least those 128 B) and the distances to the nearest parent common to the set of modified blocks. An example of the first is that the large appends and writes see only smaller benefits (11%); the number of bytes saved by avoiding copy-on-writes is small compared to the number of bytes requested to be written by the client. An example of the second is that moving a file within a directory sees greater benefit than moving a file to another directory. In the worst case, this optimization always enables the file system to avoid a copy-on-write of the inode root.

In-place append The third column (+ A or T) of Figure 3.8 shows the benefit over SP + SC of either the in-place append optimization or the independent time update optimization. In-place append enables in-place writes to data beyond the end of file (§3.3.3). The addition of this optimization immediately benefits two append operations, appending 8 B to an 8 B file and appending 4 kiB to an 8 kiB file, where it further reduces the number of bytes the file system writes by 50% and 33%, respectively. Only these two microbenchmarks see immediate benefit because they are the only microbenchmarks that append data to a file without increasing the height of the file's data tree. Though they form only a minority of our microbenchmarks, this optimization has significant impact for two reasons: additional optimizations generalize the applicability of this optimization, and appends are common in practice.

Independent time update The remaining write reductions in the third column of Figure 3.8 are due to updating access, change, and modification times independently of the commit of the associated file system operation (§3.3.4). The largest benefits occur for

System call	**SP**	**+ SC**	**+ A or T**	**+ A and T**	**+ Other**
Append					
8 B to 0 B	20,524	4,112	4,116	28	28
4 kiB to 0 B	24,612	8,200	8,204	4,116	4,116
128 kiB to 0 B	151,852	135,432	135,436	131,348	131,356
8 B to 8 B	20,532	8,200	4,112 (A)	20	20
4 kiB to 8 kiB	28,716	12,296	8,208 (A)	4,116	4,116
4 kiB to 2 MiB	24,636	8,224	8,224	8,228	4,140
128 kiB to 2 MiB	151,860	135,448	135,448	135,452	131,364
Overwrite portion of 1 MiB file					
8 B at 0 B	28,708	12,296	12 (T)	12	12
8 B at 4092 B	32,812	16,392	12,300 (T)	12,300	12,316
16 B at 0 B	28,708	12,296	4,108 (T)	4,108	4,108
4 kiB at 0 B	28,708	12,296	4,108 (T)	4,108	4,108
4 kiB at 1 B	32,812	16,392	12,300 (T)	12,300	12,316
124 kiB at 1 B	155,932	139,272	135,180 (T)	135,180	135,436
128 kiB at 0 B	155,932	139,272	135,180 (T)	135,180	135,436
Create and delete					
create	24,692	8,200	8,200	8,208	92
link	24,640	8,200	8,200	8,212	40
symlink	24,694	8,202	8,202	8,210	94
mkdir	24,698	8,202	8,202	8,210	98
unlink file	24,616	8,200	16 (T)	16	16
unlink hard link	24,624	8,200	8,200	8,212	24
unlink symlink	24,616	8,200	16 (T)	16	16
rmdir	24,620	8,200	8,200	8,208	20
Rename					
file (intra-dir.)	24,640	8,200	4,112 (T)	4,112	4,144
file (intra-dir., clobber)	24,624	8,200	4,112 (T)	4,112	4,128
file (inter-dir.)	28,752	12,296	12,296	12,312	12,360
dir. (intra-dir.)	24,644	8,200	4,116 (T)	4,116	4,148
dir. (intra-dir., clobber)	24,632	8,200	8,200	8,212	4,136
dir. (inter-dir.)	28,764	12,296	12,296	12,316	12,372
Modify attributes					
chown	20,508	4,104	12 (T)	12	12
chmod	20,504	4,104	8 (T)	8	8
read	20,500	4	4	4	4
readdir	41,000	8	8	8	8

Figure 3.8: Effectiveness of BPFS optimizations for microbenchmarks. Number of bytes written for successive optimizations. Columns are described in Section 3.6.4. Smaller numbers are better. Colors indicate percent reduction relative to the previous column: 1–24%, 25–49%, 50–89%, and 90–100%.

operations that modify only a single, 8 B region and some number of time fields. Here, the time optimization enables short-circuiting to commit the single remaining write in place. This case occurs in the following tests, where it reduces the number of bytes that BPFS writes by 99.7–99.9%: the aligned 8 B overwrite (modification time and 8 B of file data), unlinks of singly-linked files (modification time and the 8 B directory entry inode number field), the chown operation (change time, the user field, and the group field), and the chmod operation (change time and the mode field). Unlinking a file with multiple hard links does not see a benefit because it also updates the link count field in the inode; this forces a copy-on-write of the block that contains the inode. The chown operation modifies two inode fields in addition to the change time field, but each of these fields is 4 B and the two are placed in a single 8 B-aligned region. The other improved tests see a spectrum of benefit, with reductions in the range of 3–67%. These all avoid the copy-on-write for the inode block but must either copy-on-write other blocks or write to new blocks. Finally, all our tested system calls modify at least one time field, but they do not all immediately benefit from this optimization. The time update optimization provides no benefits beyond short-circuiting when the block containing the inode must still be copied for other reasons or when short-circuiting suffices because the operation modifies one time field and no other data.

In-place append and independent time update The combination of the append and time update optimizations also allows the first four append operations to avoid the copy-on-write of the inode block. These tests see additional benefit because they modify both the size and modification time fields of the inode. Results are shown in the fourth column (+ A and T) of Figure 3.8. The relative benefit is a function of the number of bytes the operation must write, and ranges from 3–99.5% for the 128 kiB append to an empty file to the 8 B append to an 8 B file. The last two append tests do not benefit because they modify the root block pointer in the inode to increase the height of the tree. Short-circuiting cannot atomically change both this field and the size of the file with a single, in-place write. Although they also change the root block pointer field, the appends to empty files see additional benefit because BPFS ignores the root block pointer field when the size of the file is 0 B.

In-place tree height switch The last column (+ Other) of Figure 3.8 combines the remaining three, more layout-specific BPFS optimizations. We discuss the results over these next three paragraphs. Adding the tree height field to inodes (§3.3.3) allows the file system to eliminate the copy-on-write for the two append microbenchmarks that did not immediately benefit from the append and time optimizations. The addition of the tree height field allows the file system to switch to a taller tree without also updating the size of the file; this is the remaining copy-on-write that is eliminated in these two microbenchmarks. This optimization further reduces the number of bytes the file system writes by 50% for the 4 kiB append and by 3% for the 128 kiB append. The 4 kiB append sees a larger relative benefit than the 128 kiB append because the overhead this optimization eliminates is fixed in size (a copy-on-write of the inode's block).

In-place file creation Enabling in-place writes to unallocated inodes and directory entries (§3.3.3) eliminates the remaining copy-on-writes for the create and symlink microbenchmarks in the last column of Figure 3.8. Although shadow paging performs a copy-on-write to modify any allocated block, BPFS exploits the finer-grained allocations within a file to safely allow in-place writes to inodes and directory entries that are unallocated. This eliminates the copy-on-writes made to initialize the inode and directory entry each create and symlink operation allocates. Eliminating these final copy-on-writes further reduces the number of bytes the file system writes by 98.9% for create and symlink.

Normalized link counts Not guaranteeing the consistency of the link count field in inodes (the inode link count optimization in §3.3.2) eliminates the remaining copy-on-writes for the unlink hard link microbenchmark in the last column of Figure 3.8. This allows the operation to unlink a hard-linked file with a single, in-place write to the directory entry's inode number field. Without this optimization the file system has to simultaneously decrement the link count in the target inode. This optimization further reduces the number of bytes the file system writes by 99.7% for unlink.

In-place file creation and normalized link counts Finally, the combination of the previous two optimizations eliminates the remaining copy-on-writes that were required for the link, mkdir, rmdir, and directory rename (intra-directory, clobber) microbenchmarks in the last column of Figure 3.8. These microbenchmarks write to unallocated directory entries and inodes and also change inode link counts. The combination of these two optimizations further reduces the number of bytes the file system writes to add and delete by 98.8–99.8% and to rename by 50%.

Macrobenchmarks

Figure 3.9 reports the number of bytes that BPFS writes for increasing levels of optimization enabled and for each of the macrobenchmarks. For each macrobenchmark the graph reports results for BPFS in shadow paging mode; with the addition of short-circuiting; with the further addition of the best of either the append or time update; then with both of the append and the time update optimizations; and, finally, also with the more layout-specific optimizations enabled.

As seen with the microbenchmarks, each optimization group significantly reduces the number of bytes the file system writes for at least some macrobenchmarks. Short-circuiting and the combination of append and time update improves this metric for all macrobenchmarks. The group of more layout-specific optimizations improves this metric for the untar, delete, and compile macrobenchmarks.

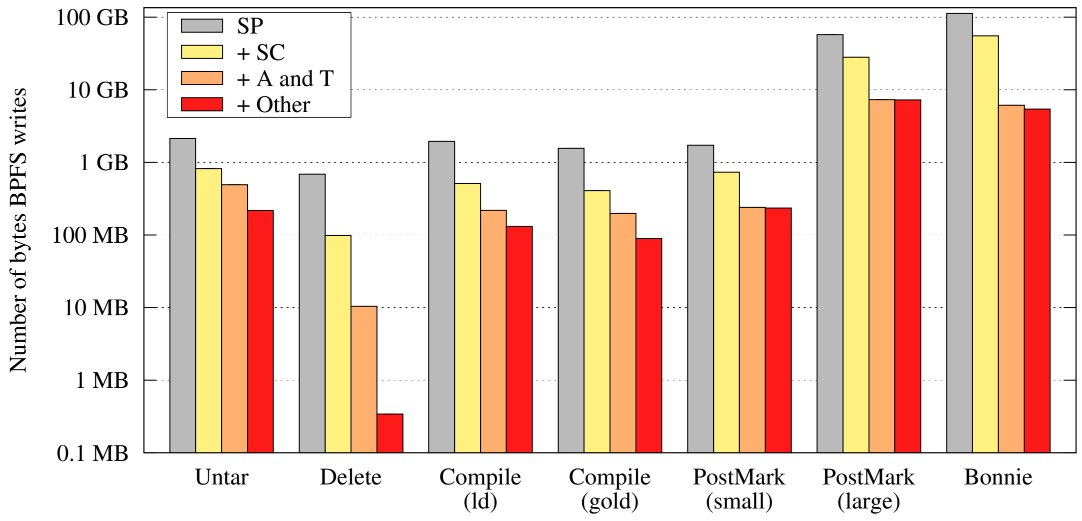


Figure 3.9: Effectiveness of BPFS optimizations for macrobenchmarks. Shorter bars are better.

Overall, compared to shadow paging, enabling all optimizations typically reduces the number of bytes the macrobenchmarks write by approximately one order of magnitude. The delete (three orders of magnitude) and Bonnie (two orders of magnitude) benchmarks see greater reductions. Contributing to this, the short-circuiting and the append and time optimizations always provide significant reductions.

3.6.5 BPFS Optimization Correctness

Whereas shadow paging writes only to unallocated blocks, the BPFS optimizations allow the file system to make many writes in place. We inspect BPFS's writes for several benchmarks to verify that, at least for these benchmarks, our optimizations maintain commit atomicity.

We want to detect any file system state that represents an atomicity violation, including states that exist only between two adjacent CPU instructions. To do this we developed a Pintool, bpfsatomic, that checks individual CPU instructions. Bpfsatomic checks that each BPRAM write instruction causes one of three file system state transitions: either the file system remains equivalent to its pre-operation state, the file system switches to an equivalent of its post-operation state, or the file system remains equivalent to its post-operation state. To compare two file system states bpfsatomic compares the hashes of their logical contents.

We test the microbenchmarks and the untar, delete, small PostMark, and compile (both ld and gold) macrobenchmarks and find that all file system operations are atomic.

We also test a random subset (1%) of the file system operations for the Bonnie macrobenchmark and find that all tested operations are atomic.

As evidence that these workloads can detect non-atomic file system operations, we note that through them bpfsatomic discovered two bugs in BPFS and that it also detects the two

Benchmark	Normalized IPC
Append	2.9
Random write	1.5
Create	1.6
PostMark	1.5

Figure 3.10: Normalized number of instructions per cycle (IPC) for BPFS-Windows with an epoch barrier cache, relative to a write-through cache. Larger numbers are better.

bugs that we intentionally insert. The first bug that bpfsatomic discovered occurred during a file create when BPFS left a field uninitialized for part of the file system operation. As part of allocating a directory entry from the unused region at the end of a directory file, BPFS changes the new directory entry's `rec_len` field from its special end-of-file value to the entry's actual length. The problem is that the non-EOF `rec_len` field value implies that the directory entry's `ino` field is valid, but BPFS left the `ino` field uninitialized until the final, committing write. We fixed this bug by setting the `ino` field to its special inactive value before setting the `rec_len` field. In the second bug that bpfsatomic discovered, BPFS used `memcpy` to commit a class of file system operations (overwrites that fit within an 8 B-aligned region). Although a commit write must be made by a single write instruction, `memcpy` can issue multiple write instructions. We fixed this bug by introducing an `atomic_memcpy`. We also inserted two bugs intentionally to test bpfsatomic. The first affects file appends. At the start of the operation it briefly changes the inode's size field to its post-operation value and then reverts it with the subsequent CPU instruction. The second intentional bug corrupts and then restores the first byte of each data block during file writes.

3.6.6 Epoch Barrier Optimization Effectiveness

In this section we measure the effectiveness of our epoch-based cache relative to using BPFS with a write-through cache. To compare the two cache systems we run BPFS in a microarchitectural simulator and measure the average number of instructions per cycle (IPC). Larger IPC values mean shorter run-times. To do this we run BPFS in the SESC simulation environment [58]. We model a dual-core 4-issue out-of-order superscalar processor with 4 MiB of L2 cache and a DDR2-800 memory system. We augment SESC's DRAM model with a command-level, DDR2-compatible PCM interface. We also modify the cache subsystem to implement epoch-based caching, including the sources of overhead mentioned in Section 3.5.4. SESC is not a full-system simulator, so cannot boot a real operating system. Therefore we run a user-level version of BPFS-Kernel as a Windows application; we call this BPFS-Windows. We ported the following four benchmarks to BPFS-Windows. Three are microbenchmarks that stress a range of file system characteristics; these are data vs. metadata updates and bulk writes vs. complex operations that require numerous ordering invariants. The fourth benchmark is the reimplementation of the PostMark macrobenchark from Section 3.6.3.

File system	**Durable**	**Atomic**			**Ordered**	
		Metadata	Data (block)	Data (syscall)	Metadata	Data (syscall)
Write-back	–	–	aligned 8 B	–	–	–
Epoch barriers	–	✓	✓	✓	intra-thread	
Write-through	✓	✓	✓	✓	✓	✓

Figure 3.11: Commit guarantees for BPFS in various caching modes. Table headings are described with Figure 3.5. More check marks means stronger guarantees.

Figure 3.10 shows the results of these experiments; results are normalized to the IPC values of the write-through scheme. All four benchmarks see a speedup of at least 49% from epoch-based caching vs. write-through caching. PostMark sees the minimum speedup, the append benchmark sees the maximum at 180%, and the average speedup is 81%.

These results show that coalescing writes and issuing them asynchronously can, compared to using a write-through cache, increase the efficiency with which software can write to PCM. Coalescing gives the on-chip cache subsystem more freedom to schedule writes. Issuing writes asynchronously uses the on-chip cache space to allow applications to process I/O-intensive code regions at processor speeds and lazily perform PCM writes during later, CPU-intensive phases of execution. However, this increase in performance comes with a reduction in the durability and ordering guarantees that BPFS provides to applications. Section 3.4.4 described these limitations; we summarize them here in Figure 3.11.

3.6.7 Summary

We have shown that BPFS writes fewer than 3 times the number that the comparison file systems write for most benchmarks. We believe this means that BPFS on PCM will generally be no slower than these comparison file systems, even though BPFS commits each operation synchronously and atomically. Existing file systems can provide neither of these guarantees at this performance. Our macrobenchark results show when BPFS incurs the largest overheads compared to existing file systems. These are when batching allows those file systems to omit intermediate writes and when BPFS incurs significant copy-on-write overheads for its atomic file writes. We have shown that each BPFS optimization is crucial for the efficiency of at least some file system operations. We have also inspected the writes for several benchmarks and verified that, at least for these benchmarks, our optimizations maintain the commit atomicity guarantees that shadow paging provides. Finally, we have shown that epoch barriers provide a performance benefit compared to write-through caching for BPFS.

3.7 Future Work

In this work we have steadfastly pushed towards optimizing the performance of the shadow paging commit mechanism towards our goals. At the same time, I think approaches focused on journaling as well as hybrids of these two may be interesting to explore. Given the byte-addressability of BPRAM, journaling file systems may no longer need to log writes at the granularity of disk sectors to ensure idempotency; additionally, journaling may be a good foundation if write ordering proves to be a significant performance factor. The two remaining sources of copy-on-write overhead for BPFS are file renames and small file overwrites. Additionally, BPFS updates timestamps independently to avoid copy-on-writes. For some of these cases, fine-grained journaling may be able to write significantly fewer bytes than the commit techniques BPFS has at its disposal.

There may be VFS interface additions that would be useful for applications and which file systems can efficiently support with BPRAM. For example, we may be able to make operations more efficient, perhaps moving data from one file into another or sharing data between two files. We may also be able to provide ways for applications to request stronger atomicity guarantees, such as atomically writing multiple regions of a file (i.e., `writev`) or of multiple files or deleting a tree of files.

3.8 Summary

BPFS is a file system designed to use byte-addressable, persistent memory with write atomicity guarantees. It stands out from other file systems for using these properties to commit each file system operation synchronously and atomically while also providing reasonable performance. We believe these robustness guarantees will make it easier to develop robust applications and allow such programs to operate more efficiently. BPFS provides these guarantees by using shadow paging and a combination of data structure design and commit optimizations to reduce the breadth of the copy-on-writes needed for a commit. For all but two system calls, `write` and `rename`, these reductions replace all copy-on-writes with in-place writes. We measure the write transfer overheads of BPFS and believe these results show that BPFS on PCM will be no slower than existing file systems on disks. Additionally, we show that each of the optimizations we have developed contribute to the overall write efficiency of BPFS.

Chapter 4

Conclusion

This dissertation improves the consistency and durability guarantees that file systems can efficiently provide.

The patch and patchgroup abstractions separate file system write ordering from durability. This allows software to specify additional ordering constraints for the file system and buffer cache to enforce. As a result, applications need not enforce these guarantees through expensive durability requests, and the file system can still improve performance by reordering and delaying disk accesses. We show how this new interface simplifies the implementation of existing consistency mechanisms like journaling and soft updates, allows applications to efficiently provide consistency guarantees where they used to provide none, and allows other applications to improve their performance while preserving their existing consistency guarantees. We make this abstraction performant using generic dependency analysis to omit unnecessary patches and patch data and to simplify patch computations.

The BPFS file system dramatically lowers the overheads of enforcing durability and consistency. BPFS uses upcoming byte-addressable, persistent memory technologies like phase change memory in place of disks and flash. We show how careful file system design for byte-addressability, improved throughput and latency, and our atomic write primitive allows BPFS to eliminate copy-on-writes that have until now been required to implement shadow paging. We use this efficiency to commit each file system operation synchronously and atomically. Our evaluation shows that, because of each optimization in the design of the file system, BPFS provides its exceptionally stronger guarantees on phase change memory without lowering the throughput that today's file systems achieve on disks.

References

[1] Apache HTTP Server. `http://httpd.apache.org/` (retrieved November 2010).

[2] Valerie Aurora. Featherstitch: Killing fsync() softly. Linux Weekly News article, September 30 2009. `http://lwn.net/Articles/354861/` (retrieved November 2010).

[3] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, M. Tosi, R. Bez, R. Gastaldi, and G. Casagrande. An 8Mb demonstrator for high-density 1.8V phase-change memories. In *Proc. 2004 Symposium on VLSI Circuits*, pages 442–445, Brianza, Italy, June 2004.

[4] Bonnie++. `http://www.coker.com.au/bonnie++/` (retrieved November 2010).

[5] Btrfs. `http://btrfs.wiki.kernel.org/` (retrieved November 2010).

[6] Bug 421482—Firefox 3 uses fsync excessively. Mozilla Bugzilla, March 7 2008. `https://bugzilla.mozilla.org/show_bug.cgi?id=421482` (retrieved November 2010).

[7] Nathan Christopher Burnett. *Information and Control in File System Buffer Management*. PhD thesis, University of Wisconsin–Madison, July 2006.

[8] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The Rio file cache: Surviving operating system crashes. In *Proc. 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, pages 74–83, Cambridge, Massachusetts, December 1996.

[9] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, Big Sky, Montana, October 2009.

[10] M. Crispin. Internet Message Access Protocol—version 4rev1 (IMAP). RFC 3501, Internet Engineering Task Force, March 2003. `http://tools.ietf.org/html/rfc3501` (retrieved November 2010).

[11] Dovecot, Version 1.0 beta7, April 2006. `http://www.dovecot.org/` (retrieved October 2010).

[12] International Technology Roadmap for Semiconductors (ITRS). Process integration, devices, and structures, 2007. `http://www.itrs.net/links/2007itrs/2007_chapters/2007_PIDS.pdf` (retrieved November 2010).

[13] FreeBSD. *gjournal—control utility for journaled devices*. `http://www.freebsd.org/cgi/man.cgi?query=gjournal` (retrieved October 2010).

[14] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, pages 307–320, Stevenson, Washington, October 2007.

[15] FUSE (Filesystem in Userspace). `http://fuse.sourceforge.net/` (retrieved November 2010).

[16] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *Proc. 2005 USENIX Annual Technical Conference*, pages 89–104, Anaheim, California, April 2005.

[17] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Transactions on Computer Systems (TOCS)*, 18(2):127–153, May 2000.

[18] gzip. `http://www.gzip.org/` (retrieved November 2010).

[19] Robert Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proc. 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 155–162, Austin, Texas, November 1987.

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

[21] Dave Hitz, James Lau, and Michael Malcolm. File system design for an NFS file server appliance. In *Proc. USENIX Winter 1994 Technical Conference*, pages 235–246, San Francisco, California, January 1994.

[22] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, February 1988.

[23] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 263–276, Brighton, England, October 2005.

[24] Intel. *Product Manual: Intel X25-E Extreme SATA Solid State Drive*, May 2009. `http://download.intel.com/design/flash/nand/extreme/319984.pdf` (retrieved November 2010).

[25] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, 253668-035US edition, September 2010. `http://www.intel.com/Assets/PDF/manual/253668.pdf` (retrieved November 2010).

[26] Guillem Jover. ext4 and data consistency with dpkg. Linux Weekly News comment, June 18 2010. `http://lwn.net/Articles/392599/` (retrieved November 2010).

[27] Guillem Jover. Bits from the dpkg team. Debian devel announce list, March 26 2010. `http://lwn.net/Articles/380941/` (retrieved November 2010).

[28] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on Exokernel systems. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.

[29] Jeffrey Katcher. PostMark: A new file system benchmark. Technical Report TR3022, Network Appliance, 1997. `http://web.archive.org/web/20050901112245/http://www.netapp.com/tech_library/3022.html` (retrieved November 2010).

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

[31] Alexander Larsson. Bug 575555—use fsync() when replacing files to avoid data loss on crash. GNOME Bugzilla, March 16 2009. `https://bugzilla.gnome.org/show_bug.cgi?id=575555` (retrieved November 2010).

[32] Alexander Larsson. fsync in glib/gio. GTK development list, March 12 2009. `http://mail.gnome.org/archives/gtk-devel-list/2009-March/msg00082.html` (retrieved November 2010).

[33] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proc. 36th IEEE International Symposium on Computer Architecture (ISCA '09)*, pages 2–13, Austin, Texas, June 2009.

[34] Linux kernel, Version 2.6.34.1, July 2010. `http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.34.1.tar.bz2` (retrieved November 2010).

[35] Barbara Liskov and Rodrigo Rodrigues. Transactional file systems can be fast. In *Proc. 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September 2004.

[36] David E. Lowell and Peter M. Chen. Free transactions with Rio Vista. In *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 92–101, Saint-Malô, France, October 1997.

[37] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 190–200, Chicago, Illinois, June 2005.

[38] Phu Ly. Fsyncers and curveballs. Phu Ly's blog, May 25 2008. `http://shaver.off.net/diary/2008/05/25/fsyncers-and-curveballs/` (retrieved November 2010).

[39] Michael D. Mammarella. *Data Storage Considered Modular*. PhD thesis, University of California, Los Angeles, February 2010.

[40] Timothy Mann, Andrew Birrell, Andy Hisgen, Charles Jerian, and Garret Swart. A coherent distributed file cache with directory write-behind. *ACM Transactions on Computer Systems (TOCS)*, 12(2):123–164, May 1994.

[41] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems (TOCS)*, 2(3): 181–197, August 1984.

[42] Marshall Kirk McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the Fast Filesystem. In *Proc. 1999 USENIX Annual Technical Conference, FREENIX Track*, pages 1–17, Monterey, California, June 1999.

[43] R. Micheloni, A. Marelli, and R. Ravasio. BCH hardware implementation in NAND Flash memories. In *Error Correction Codes in Non-Volatile Memories*. Springer Netherlands, 2008.

[44] Jeffrey C. Mogul, Eduardo Argollo, Mehul Shah, and Paolo Faraboschi. Operating system support for NVM+DRAM hybrid main memory. In *Proc. 12th Hot Topics in Operating Systems Symposium (HotOS-XII)*, Monte Verità, Switzerland, May 2009.

[45] C. Mohan. Repeating history beyond ARIES. In *Proc. 25th International Conference on Very Large Data Bases (VLDB '99)*, Edinburgh, Scotland, September 1999.

[46] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems (TODS)*, 17(1):94–162, 1992.

[47] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 191–205, Brighton, England, October 2005.

[48] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 1–14, Seattle, Washington, November 2006.

[49] University of Washington. UW IMAP toolkit. `http://www.washington.edu/imap/` (retrieved November 2010).

[50] Oracle. ZFS. `http://hub.opensolaris.org/bin/view/Community+Group+zfs/WebHome` (retrieved November 2010).

[51] Oracle. OCFS2. `http://oss.oracle.com/projects/ocfs2/` (retrieved November 2010).

[52] Mondira Deb Pant, Pankaj Pant, and Donald Scott Wills. On-chip decoupling capacitor optimization using architectural level prediction. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 10(3):319–326, 2002.

[53] Neil Perrin. Re: [zfs-discuss] is write(2) made durable atomically? ZFS-discuss mailing list, November 30 2009. `http://www.mail-archive.com/zfs-discuss@opensolaris.org/msg31651.html` (retrieved November 2010).

[54] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, England, October 2005.

[55] Qualitative Software. RAMDisk. `http://www.ramdisk.tk/` (retrieved November 2010).

[56] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. 36th IEEE International Symposium on Computer Architecture (ISCA '09)*, pages 24–33, Austin, Texas, June 2009.

[57] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4):465–479, July 2008.

[58] Jose Renau, Basilio Fraguela, James Tuck, Wei Liu, Milos Prvulovic, Luis Ceze, Smruti Sarangi, Paul Sack, Karin Strauss, and Pablo Montesinos. SESC simulator, January 2005. `http://sesc.sourceforge.net` (retrieved November 2010).

[59] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1): 26–52, 1992.

[60] David S. H. Rosenthal. Evolving the Vnode interface. In *Proc. USENIX Summer 1990 Technical Conference*, pages 107–118, Anaheim, California, June 1990.

[61] Malcom Rowe. Re: wc atomic rename safety on non-ext3 file systems. Subversion developer mailing list, March 5 2007. `http://svn.haxx.se/dev/archive-2007-03/0064.shtml` (retrieved November 2010).

[62] Margo I. Seltzer, Gregory R. Ganger, Marshall K. McKusick, Keith A. Smith, Craig A. N. Soules, and Christopher A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proc. 2000 USENIX Annual Technical Conference*, pages 71–84, San Diego, California, June 2000.

[63] Gopalan Sivathanu, Swaminathan Sundararaman, and Erez Zadok. Type-safe disks. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 15–28, Seattle, Washington, November 2006.

[64] Glenn C. Skinner and Thomas K. Wong. "Stacking" Vnodes: A progress report. In *Proc. USENIX Summer 1993 Technical Conference*, pages 161–174, Cincinnati, Ohio, June 1993.

[65] Larry D. Smith, Raymond E. Anderson, Doug W. Forehand, Thomas J. Pelc, and Tanmoy Roy. Power distribution system design methodology and capacitor selection for modern CMOS technology. *IEEE Transactions on Advanced Packaging*, 22(3): 284–291, August 1999.

[66] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling transactional file access via lightweight kernel extensons. In *Proc. 7th USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, California, February 2009.

[67] Subversion. `http://subversion.apache.org/` (retrieved November 2010).

[68] subversion: Issue 3015. Subversion issue tracker, November 6 2007. `http://subversion.tigris.org/issues/show_bug.cgi?id=3015` (retrieved November 2010).

[69] subversion: Issue 628. Subversion issue tracker, February 8 2002. `http://subversion.tigris.org/issues/show_bug.cgi?id=628` (retrieved November 2010).

[70] Theodore Ts'o. Re: [evals] ext3 vs reiser with quotas. Mailing list, December 19 2004. `http://linuxmafia.com/faq/Filesystems/reiserfs.html` (retrieved November 2010).

[71] Theodore Ts'o. Ubuntu bug #317781: Ext4 data loss, comment #45. Ubuntu bug tracker, March 6 2009. `https://bugs.edge.launchpad.net/ubuntu/+source/linux/+bug/317781/comments/45` (retrieved November 2010).

[72] Theodore Ts'o. Ubuntu bug #317781: Ext4 data loss, comment #54. Ubuntu bug tracker, March 7 2009. `https://bugs.edge.launchpad.net/ubuntu/+source/linux/+bug/317781/comments/54` (retrieved November 2010).

[73] Theodore Ts'o. Delayed allocation and the zero-length file problem. Theodore Ts'o's blog, March 15 2009. `http://thunk.org/tytso/blog/2009/03/15/dont-fear-the-fsync/` (retrieved November 2010).

[74] Theodore Ts'o. Delayed allocation and the zero-length file problem, comment #39. Theodore Ts'o's blog, March 15 2009. http://thunk.org/tytso/blog/2009/03/12/delayed-allocation-and-the-zero-length-file-problem/#comment-2021 (retrieved November 2010).

[75] Theodore Ts'o. Don't fear the fsync! Theodore Ts'o's blog, March 15 2009. http://thunk.org/tytso/blog/2009/03/15/dont-fear-the-fsync/ (retrieved November 2010).

[76] Theodore Ts'o. btrfs fscked up, too? Linux Weekly News comment, March 16 2010. http://lwn.net/Articles/323745/ (retrieved November 2010).

[77] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *Proc. 4th Annual LinuxExpo*, Durham, North Carolina, May 1998.

[78] Ubuntu bug #317781: Ext4 data loss. Ubuntu bug tracker, January 2009. https://bugs.launchpad.net/ubuntu/+source/linux/+bug/317781 (retrieved November 2010).

[79] Steve VanDeBogart, Christopher Frost, and Eddie Kohler. Reducing seek overhead with application-directed prefetching. In *Proc. 2009 USENIX Annual Technical Conference*, pages 299–312, San Diego, California, June 2009.

[80] Murali Vilayannur, Partho Nath, and Anand Sivasubramaniam. Providing tunable consistency for a parallel file store. In *Proc. 4th USENIX Conference on File and Storage Technologies (FAST '05)*, pages 17–30, San Francisco, California, December 2005.

[81] An-I Andy Wang, Peter Reiher, Gerald J. Popek, and Geoffrey H. Kuenning. Conquest: Better performance through a disk/persistent-RAM hybrid file system. In *Proc. 2002 USENIX Annual Technical Conference*, pages 15–28, Monterey, California, June 2002.

[82] Mike Waychison. Re: fallocate support for bitmap-based files. linux-ext4 mailing list, June 29 2007. http://www.mail-archive.com/linux-ext4@vger.kernel.org/msg02382.html (retrieved November 2010).

[83] Ralph O. Weber. *SCSI Architecture Model - 3 (SAM-3)*. Accredited Standards Committee INCITS, Technical Committee T10, revision 14 edition, September 21 2004. `http://www.t10.org/ftp/t10/drafts/sam3/sam3r14.pdf`.

[84] Ralph O. Weber. *SCSI Primary Commands - 4 (SPC-4)*. Accredited Standards Committee INCITS, Technical Committee T10, revision 18 edition, February 18 2009. `http://www.t10.org/cgi-bin/ac.pl?t=f&f=spc4r18.pdf`.

[85] David Woodhouse. JFFS: The journalling flash file system. In *Proc. 2001 Ottawa Linux Symposium*, Ottawa, Canada, July 2001. `http://sources.redhat.com/jffs2/jffs2.pdf` (retrieved November 2010).

[86] Charles P. Wright, Michael C. Martino, and Erez Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proc. 2003 USENIX Annual Technical Conference*, pages 197–210, San Antonio, Texas, June 2003.

[87] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID semantics to the file system. *ACM Transactions on Storage*, 3(2):1–40, June 2007.

[88] Michael Wu and Willy Zwaenepoel. eNVy: A non-volatile, main memory storage system. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 86–97, San Jose, California, October 1994.

[89] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathni. Using model checking to find serious file system errors. In *Proc. 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, San Francisco, California, December 2004.

[90] Erez Zadok and Jason Nieh. FiST: A language for stackable file systems. In *Proc. 2000 USENIX Annual Technical Conference*, pages 55–70, San Diego, California, June 2000.

[91] Erez Zadok, Ion Badulescu, and Alex Shender. Extending file systems using stackable templates. In *Proc. 1999 USENIX Annual Technical Conference*, pages 57–70, Monterey, California, June 1999.

[92] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proc. 36th IEEE International Symposium on Computer Architecture (ISCA '09)*, pages 14–23, Austin, Texas, June 2009.