Building a File-Based Storage Stack: Modularity and Flexibility in Loris

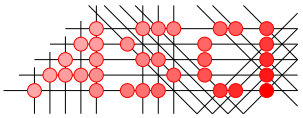
Ph.D. Thesis

Raja Appuswamy

VU University Amsterdam, 2014

vrije Universiteit *amsterdam*

This work was supported by the European Research Council Advanced Grant 227874.



This work was carried out in the ASCI graduate school.
ASCI dissertation series number 307.

ISBN 978-90-5383-087-1

Printed by Wöhrmann Print Service

VRIJE UNIVERSITEIT

BUILDING A FILE-BASED STORAGE STACK: MODULARITY AND FLEXIBILITY IN LORIS

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op maandag 2 juni 2014 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

Raja Appuswamy

geboren te Chennai, Tamilnadu, India

promotor: prof.dr. A.S. Tanenbaum

“Today, most software exists, not to solve a problem,
but to interface with other software”

I.O.Angell

Table of Contents

ACKNOWLEDGEMENTS **13**

1 Introduction **17**
1.1 Evolution of File and Storage Systems 18
1.1.1 File Systems 18
1.1.2 Logical Addressing and RAID 19
1.1.3 Volume Management 20
1.1.4 User-Level Metadata Management 21
1.1.5 Second-Level Flash Integration 21
1.2 Retiring the Traditional Stack 22
1.3 Introducing Loris 23
1.4 Contributions of this Thesis 25
1.5 Organization of this Thesis 26

2 Loris **29**
2.1 Introduction 30
2.2 Problems with the Traditional Storage Stack 31
2.2.1 Reliability 31
2.2.2 Flexibility 33
2.2.3 Heterogeneity Issues 34
2.3 Solutions Proposed in the Literature 35
2.4 The Design of Loris 36
2.4.1 The Physical Layer 37
2.4.2 The Logical Layer 39
2.4.3 The Cache Layer 41
2.4.4 The Naming Layer 42
2.5 The Advantages of Loris 43
2.5.1 Reliability 43
2.5.2 Flexibility 44
2.5.3 Heterogeneity 45
2.6 Evaluation 45

2.6.1 Test Setup 46
2.6.2 Evaluating Reliability and Availability 46
2.6.3 Performance Evaluation 49
2.7 Conclusion 53

3 Volume Management 55
3.1 Introduction 56
3.2 Problems with existing approaches 57
3.2.1 Lack of Flexibility 57
3.2.2 Lack of support for heterogeneous devices 58
3.3 The Loris storage stack 60
3.3.1 Physical layer 60
3.3.2 Logical layer 61
3.3.3 Cache and naming layers 61
3.4 File volume virtualization in Loris 62
3.4.1 File pools: Our new storage model 62
3.4.2 Infrastructure support for file pools 63
3.4.3 Infrastructure support for file volume virtualization 64
3.5 New functionality: file volume snapshoting in Loris 65
3.5.1 Division of labor 66
3.5.2 Physical layer(1): Copy-based snapshoting 66
3.5.3 Physical layer(2): Copy-on-write-based snapshoting 67
3.5.4 File volume snapshoting in the logical layer 71
3.6 New functionality: Unifying file snapshoting and version creation policies 74
3.6.1 Version volumes 75
3.6.2 Open-close versioning in the naming layer 76
3.7 New functionality: Version directories– a unified interface for browsing history 76
3.7.1 Version directories – interface specification 77
3.7.2 Version directories – implementation details 78
3.8 Evaluation 80
3.8.1 Test Setup 80
3.8.2 Copy-based and copy-on-write snapshoting comparison . . 80
3.8.3 Open-close versioning evaluation 81
3.8.4 Overhead of file volume virtualization 81
3.9 Comparison with other approaches 83
3.9.1 Device management 83
3.9.2 File management and file volume virtualization 84
3.10 Future work 85
3.10.1 Flexible cloning in Loris 85
3.10.2 Hybrid file pools 85

3.11 Conclusion 86

4 Metadata Management 87
4.1 Introduction 88
4.2 Background: The Loris Storage Stack 90
4.2.1 Physical layer 91
4.2.2 Logical layer 92
4.2.3 Cache layer 93
4.2.4 Naming layer 93
4.3 Efficient, Modular Metadata management with Loris 93
4.3.1 Storage management sublayer 94
4.3.2 Interface management sublayer 96
4.4 Evaluation 102
4.4.1 Test setup 102
4.4.2 Microbenchmarks 103
4.4.3 Macrobenchmarks 104
4.4.4 Attribute indexing overhead 105
4.5 Related Work 106
4.5.1 Storage management 106
4.5.2 Interface management 107
4.5.3 End-to-end metadata management 107
4.6 Future Work 107
4.6.1 Partitioning 108
4.6.2 Exploiting heterogeneity 108
4.7 Conclusion 108

5 Hybrid Storage 109
5.1 Introduction 110
5.2 Hybrid storage systems 111
5.2.1 Caching 111
5.2.2 Dynamic Storage Tiering 112
5.3 Background: The Loris storage stack 113
5.3.1 Physical layer 113
5.3.2 Logical layer 113
5.3.3 Cache and Naming layers 115
5.3.4 Tiering Framework 115
5.3.5 Loris as a platform for storage tiering - The Pros 117
5.3.6 Loris as a platform for storage tiering - The Cons 118
5.4 Loris-based hybrid systems 118
5.4.1 Loris-based Hot-DST systems 119
5.4.2 Loris-based Cold-DST architectures 120
5.4.3 Loris-based Caching 121

5.5 Evaluation . . . 121
5.5.1 Test Setup . . . 122
5.5.2 Benchmarks and Workload Generators . . . 122
5.5.3 Workload Categories . . . 123
5.5.4 Comparative evaluation . . . 124
5.5.5 Mixed Workloads and Hybrid Architectures . . . 134
5.6 Discussion . . . 135
5.6.1 Analyzing Cold-DST . . . 136
5.6.2 Other hybrid architectures . . . 136
5.6.3 Caching vs Tiering Algorithms . . . 137
5.7 Conclusion . . . 137

6 Host-Side Caching **139**
6.1 Introduction . . . 140
6.1.1 Consistent, Block-Level Write-Back Caching . . . 141
6.1.2 Filesystem-Based Caching . . . 142
6.1.3 Our Contributions . . . 143
6.2 The Case For File-level Host-Side Caching With Loris . . . 144
6.2.1 Loris - Background . . . 144
6.2.2 File-level Host-side Caching With Loris . . . 147
6.3 Loris-based Host-side Cache: Architecture . . . 150
6.3.1 Volume Management Sublayer: Subfile Mapping . . . 150
6.3.2 Cache Management Sublayer . . . 152
6.3.3 Physical Layer Support For Subfile Caching . . . 152
6.3.4 Network File Store . . . 153
6.4 Evaluation . . . 153
6.4.1 Setup . . . 153
6.4.2 Benchmarks and Workload Generators . . . 154
6.4.3 Comparative Evaluation: Caching Policies . . . 154
6.4.4 Network Performance Sensitivity . . . 155
6.4.5 Cache Size Sensitivity . . . 157
6.4.6 File-Level and Block-Level Caching Comparison . . . 158
6.5 Conclusion . . . 160

7 Discussion **161**
7.1 Metadata Management . . . 161
7.1.1 Metadata Consistency . . . 161
7.1.2 Metadata Reliability . . . 163
7.2 Flash Management . . . 164
7.2.1 Tiering . . . 164
7.2.2 Caching . . . 165

8 Summary and Conclusion **167**
8.1 Summary 167
8.2 Future Work 169
8.2.1 Alternative Storage Stacks 169
8.2.2 Integrating New Storage Protocols 170
8.2.3 Extensions to Existing Protocols 171

REFERENCES **175**

SAMENVATTING **185**

Acknowledgements

It was just another afternoon in the ever-busy Microsoft Redmond campus. As I sat in my office that day, contemplating rewriting code I had written after having a heavy lunch, I received a phone call that would change my life forever. “Do you promise to work hard?” asked Andy. At that moment, I did not think about asking Andy to quantify the word “hard”. I did not pause even for a second to consider that I would be taking a pay cut, giving up an amazing job, or worst of all, selling of my brand new Bimmer. It did not bother me that I would be uprooting my stable, well-established life only to start over again as a Ph.D student on a one-year, probationary contract. My mind was preoccupied with a single thought–“I am going to be Andy Tanenbaum’s Ph.D student!”. Without hesitating, I said “Yes. I promise.”, and grabbed what I will always consider to be an opportunity of a life time.

This thesis marks the culmination of a journey, one that lasted five wonderful years since that day, and one which would have been impossible without the guidance and support offered by several amazing people. First and foremost, I would like to thank my supervisor Andy Tanenbaum. During the initial stages of my research, you patiently withstood a barrage of random ideas, taught me the value of being critical, and helped me steer clear of several research topics that have a low impact–investment ratio. As I gained experience, you gave me full autonomy and encouraged me to work on topics I was passionate about. You spent countless hours, quite often burning the midnight oil before deadlines, teaching me the art of writing good research papers. You were incredibly understanding even when times were tough and I had to make multiple, month-long, personal trips to India. You were even the source of inspiration for some of my new hobbies, be it travel photography or culture tourism. I am deeply indebted to you for all of this and much more.

Next, I would like to thank the members of my thesis committee, André Brinkmann, Ethan Miller, Raju Rangaswami, Eno Thereska, and Spyros Voulgaris, for taking the time to review this thesis. Your invaluable feedback greatly helped in improving the overall quality of this dissertation. It is an absolute honour to have such leading researchers on my committee.

I am extremely grateful to Antony Rowstron for inviting me not once, but twice, to work with a wonderful group of researchers at the Microsoft Research, Cambridge. I would like to thank all the members of the Systems and Networking Group, particularly Christos Gkantsidis, Orion Hodson, and Dushyanth Narayanan, for providing a challenging, yet truly enjoyable, research environment.

I owe a huge debt of gratitude to my paranymphs David van Moolenbroek and Cristiano Giuffrida. David, I would have definitely not accomplished as much as I did without your collaboration. During the course of my Ph.D., you have donned several roles–as a team mate spending hours on the white board brainstorming ideas, as a translator helping me make sense of all those letters from the IND, as a developer fixing bugs in quick-and-dirty prototypes I wrote, and as a friend who helped me convince the Dutch police that I was indeed a victim of "skimming"-based cybercrime. Thank you for all those memories, and I hope that I will be able to repay back the debt I owe you in the near future, when it will be your turn to face the guillotine.

Cristiano (or should I say Dr. Giuffrida), we were certainly two peas in a pod, sharing much more than just an appreciation for the fine art of dining. We started our Ph.D. on the same day, went through the peaks and troughs of our scholarly careers together, and decided that we had milked the hypothetical research cow dry more or less at the same time. I will always cherish the moments we have had, be it our long-lasting, Wednesday-night ritual of getting irritated watching "Lost" while eating pizza, or the one too many black outs we have had in the gym trying to shed the post-Christmas "muscle" mass. I wish you and Laura the very best in all your future endeavours.

I would like to thank all past and present members of the MINIX team. Jorrit Herder, Dirk Vogt, Erik van der Kouwe, and Tomas Hruby, my P4.69 office mates, and my fellow Ph.D. students, it has been a real pleasure working side by side with all of you. I have learned a great deal about several topics, ranging from optimizing multiserver systems to the classification of cannibalism among various animal species, through the interactions we have had during the all-too-frequent coffee breaks. Erik, many thanks for writing the Dutch summary of this thesis and for doubling in role as my tax advisor, financial assistant, and stand-in translator when David was not available. A special note of thanks to Lorenzo Cavallaro, for inspiring all of us with his voracious appetite for both good food and good security research. Philip Homburg, Thomas Veerman, Arun Thomas, Lionel Sambuc, and Kees Jongenburger, my fellow MINIX programmers, thank you for making me feel at home in Amsterdam. You guys have always been forthcoming and helpful in times of need. Philip, I certainly look forward to the party you will be hosting sometime in the future to celebrate the restoration of your house back to living conditions. Thomas, I cannot wait to see how you and Saskia handle those two little treasure troves ten years from now, and Saskia, I still owe you a

home-cooked Indian dinner! Arun, Lisa, I really enjoyed your company while you were here in Amsterdam and I hope that our paths cross again someday in the future. Ben, your company has been nothing short of legen–wait for it–dary, legendary. I will always remember you as that content-addressed-storage-obsessed, computer-vision-loving, playbook-owning, tweed-jacket-wearing, heavy-weight lifting, boxing champion who also writes kick-ass code. Lionel and Kees, I have always admired the passion you share for not just the MINIX project, but also for all things embedded. I will fondly remember the very many social events we have had over the last few years.

I would like to acknowledge all my other colleagues from the Computer Systems group at the Vrije Universiteit, Amsterdam, who made my doctoral life much more enjoyable, particularly Ana Oprescu for just being the ever-cheerful herself, Albana Gaba for all the tips and tricks she shared with me based on her experience dealing with several work and non-work-related issues alike, and Asia Slowinska for her stories about amazing adventures in the exotic Indian subcontinent. I would also like to thank Richard van Heueven and Sharan Santhanam, for their contributions to the Loris project. It has been a pleasure mentoring both of you. A special word of thanks to Caroline Waij and Ilse Thomson for just being the best secretaries in the history of the department (at least the history as I know it).

Despite all the support I got from my colleagues in Amsterdam, I would have never managed to finish my Ph.D. with even half a head full of hair had it not been for the morale-boosting distractions created by my friends. Rosa Meijer and Johannes Bertens, thank you for all those fun trips to beaches and amusement parks. I am super excited and looking forward to finishing our Battlestar Galactica marathon in your lovely new home. Viktorian Miok, Gwénaël Leday, and Andrea Contretras, I look forward to many more relaxing evenings, catching up and dining out amidst vibrant tourists in the Amsterdam city center. Jakub Pecanka and Melania Calinescu, thank you for so many fun-filled, board-game evenings and movie nights. Gwenny Sitters, thank you for giving me the opportunity, on more than one occasion, to enjoy the treasured company of all your lovely dogs and cats. Shankar Gnanasekharan, thank you for accommodating me in your place during my first week in Amsterdam and teaching me basic pattern matching skills that I so desperately needed to shop in Dutch supermarkets. Srijith K. Nair, thank you for all your help, assistance, and advice during the first few weeks of my stay in Amsterdam.

A big thank you to all my University of Florida friends. Amarnath Raghunathan, Jayanath Natarajan, and Raghav Panchapakesan, I would have never been here had it not been for you guys. It was your encouragement and nudging that convinced me to send that first email to Andy enquiring about an open position in his group. Preethi Prasad, Deepa Jayanth, Nithya Vijayaraghavan, Sriram Parthasarathy, Ritu Manjunathan, Kannan Rajah, Madhu Kallazhi, and Praveen Kumar Subramanian, thank you for generously hosting me each and every single

time I visited California. The only thing I regret about living in Europe is not being able to enjoy all your company as much as I would like to. A special note of thanks to Dr. Alain Anyouzoa, my former colleague at Microsoft, for constantly encouraging me to pursue a Ph.D, and for gifting me his book on UNIX system design as a memento.

Now, I would like to thank my family for their unwavering love and support. Thank you dad, for all those countless nights you stayed up with me helping me prepare for exams, for all those countless hours you spent in temples praying for my well-being, for sacrificing your career to give me a good upbringing, and for making sure that I had everything I ever wanted by using your accounting skills to balance household expenses against the only source of income–your pension. Thank you mom, for keeping me healthy and well fed even if it meant spending hours at the kitchen enduring crippling arthritic pain, for never saying no to anything I have ever asked of you, and for spending every single moment of your life taking care of the family without expecting anything in return. A big thanks to my maternal uncles, Kumar, Srinivasan, Rajasekar, aunts, Alli, and Meenakshi, and my cousin sister Nirmala. Thank you all for shouldering my responsibilities in my absence without any strings attached. Thank you for encouraging me to follow my heart even if my decisions ended up adding more work to your plate. Thank you for always being there for me since my childhood and for always treating me like a son rather than a nephew.

Before I wrap up this section, there is still one person who I need to acknowledge. The only problem is that the list of things this person needs to be thanked for is so long that I would have to devote the next three pages, should I decide to write it all down. When I started making an outline for these three pages, I realized that I would pretty much be repeating the contents of the first three pages with all references replaced by a single name–Nimisha Chaturvedi. My beloved wife, you are my everything–my mentor who guides me in making the right personal and professional choices, my colleague who understands the deadline-driven academic lifestyle and even proof reads my research papers, my best friend who always finds a way to cheer me up even in the toughest of times, my roommate who cooks mind-blowing food without having me move a muscle, and my family member who introduced me to the world's best parents-in-law, Dr. Anoop Chaturvedi and Mrs. Sunita Chaturvedi, and the best brother-in-law ever, the one and only Apoorv Chaturvedi. Thank you mom and dad, for your trust, your support, your words of wisdom, and your love. Thank you Apoorv, for always taking my side in debates with Nimisha, and more importantly, for being the brother who is always looking for ways to make my life easier. Thank you Nimisha, my love, for giving me a lifetime of happiness.

Raja Appuswamy
Amsterdam, The Netherlands, April 2014

Chapter 1

General Introduction

The two major factors that influence the design of modern computer systems are undoubtedly changes in application requirements and advances in hardware technologies. File and storage systems are no different. Originally used for providing online storage to users in multiprogramming environments, file systems are now used in a dizzying array of application domains for providing application-independent persistent data storage. Traditionally designed to manage direct-attached, block-based, magnetic storage media, storage systems of today interface with a wide range of devices with radically different interfaces and price/performance/reliability tradeoffs. To accommodate these changes, storage software has also evolved into a complex, multilayered collection of protocols that run in different contexts (user applications, operating system modules, and firmware).

Layering and abstractions are the tools of trade employed by computer systems designers for building modular software. For instance, the network stack in modern operating systems consists of several layers with a well-defined division of labor. Similar to the network stack, the multilayered collection of storage protocols has been referred to as the *storage stack* [24], as these layers also communicate using standardized interfaces that offer well-defined abstraction boundaries. In both protocol stacks, the strict separation of layers provided by these abstract interfaces enables the addition of new protocols, and updates to existing protocols, in one layer without affecting others. However, unlike the systematic, standardized evolution of protocol layering in the network stack, layering in the storage stack was driven by one factor—compatibility with existing storage installations.

In this thesis, we will show how this compatibility-driven protocol layering in the traditional stack causes various problems that render it ineffective in managing modern-day storage installations and incapable of accommodating future changes in the storage hardware landscape. In doing so, we will present a new layering of storage protocols which sacrifices backwards compatibility in favor of modularity, and show how the resulting storage stack, which we refer to as Loris, solves all the

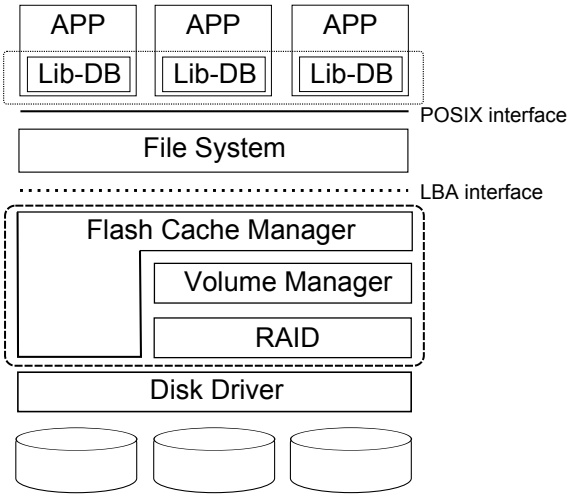


Figure 1.1: The figure depicts the layering of protocols in the traditional storage stack. The dotted line represents the transition from file to block level. The protocols sandwiched within the dashed rectangle collective form the RAID layer, and those within the dotted rectangle form the Metadata Management layer.

issues that plague the traditional layering by design. But first, we will set the stage for our research by presenting a brief account of the evolution of the traditional storage stack.

1.1 Evolution of File and Storage Systems

Figure 1.1 shows the protocol layering in the storage stack found in most operating systems. This multilayered collection of protocols, formed as a result of decades of evolution, had humble beginnings—a single file system layer. We will now trace the evolution of the stack starting with first-generation file systems and show how backwards compatibility played a pivotal role in shaping the protocol layering seen today.

1.1.1 File Systems

The task of first-generation file systems, like the Multics file system [23], was to provide a general-purpose, device-independent mechanism for addressing a large amount of direct-attached secondary storage. In order to achieve this objective, the Multics file system provided two abstractions which user-space applications could use to store and retrieve data in a device-independent fashion, namely, files and directories. A file was defined to be a symbolically named, ordered sequence

of elements, which, from the file system point of view, was format free. A directory, on the other hand, was a special file maintained by the file system and contained a list of entries, which, to a user, appears to be files that can be accessed using their symbolic names. Using these two abstractions, the Multics file system demonstrated the utility of hierarchical file management, which, until now, has remained the primary mechanism of interfacing with file systems.

Following the Multics file system, the UNIX file system [78] implemented and extended the file abstraction to I/O devices. The UNIX file system also introduced a simple, yet efficient, on-disk layout in which files were represented using inodes that stored pointers to data blocks in addition to other POSIX metadata. However, the simplicity of the UNIX file system became its Achilles' heel, as its design failed to exploit the increasing bandwidth of hard disk drives due to the random accesses caused by suboptimal allocation of data blocks and strict separation of metadata and data.

Although a wide variety of techniques have been proposed in literature for overcoming the performance bottleneck caused by the UNIX on-disk layout, the approach taken by Berkeley Fast File System (FFS) [63] deserves a special mention. FFS improved performance significantly using several optimizations, some of which exploited the internal geometry details of hard disk drives, exposed by the then-prevalent CHS (Cylinder, Head, Sector) storage interface, to perform allocation of blocks belonging to the same file within the same cylinder group at rotationally-optimal positions. Thus, file systems were not only aware of internal device geometry, but were designed based on the assumption that storage devices were physically addressed and direct attached.

1.1.2 Logical Addressing and RAID

The SCSI standard of 1986 invalidated the assumptions made by file systems like FFS. The standard replaced the physical CHS interface with a simple Logical Block Address (LBA) interface that abstracted away device geometry information, enabling the development of interoperable, device-independent systems and peripherals. The LBA interface is arguably one of the most stable interfaces in the history of computer systems as it continues to remain the dominating way of accessing storage devices.

The abstract view of storage provided by the LBA interface fostered innovations above and below the interface. File systems evolved rapidly in an effort to close the ever-increasing CPU–storage bottleneck using various techniques like optimizing reads using clustered data allocation [64], accelerating writes using log-structured storage layout [80], etcetera. New techniques, like journaling [39], soft updates [29], and shadow paging [45] were also developed to improve file system reliability in the face of crashes and power failures. Despite these advances, file systems continued to be designed based on the "one-file system-per-

disk" bond until it was realized that "Single-Large-Expensive-Disks" (SLEDs) could be replaced by a "Redundant-Array-of-Inexpensive-Disks" (RAID) to improve both performance and reliability.

The advent of RAID [73] techniques is a landmark in the history of storage systems, as it introduced a new layer, which we will henceforth refer to as the *RAID Layer*, in the file system-based storage stack. The LBA interface played a pivotal role in the formation of this new layer as it enabled RAID algorithms to be integrated at the block level, in hardware or software, thereby retaining compatibility with existing file systems. Thus, RAID implementations were able to hide their internal complexity by presenting a logical disk to the file systems: a linear array of blocks that can be read and written.

The disk array industry is today a multibillion-dollar industry and redundant disk arrays have become the dominant form of storage for many high-end installations. Although the traditional RAID levels have been extended to support multiple disk failures using sophisticated erasure codes [13; 22], these new redundancy techniques continue to be integrated into the RAID layer.

1.1.3 Volume Management

As storage installations grew in size, storage administrators resorted to using file systems as basic administrative units for grouping related data and enforcing various policies, like setting storage quotas, access permissions etcetera. With the addition of the RAID layer, file systems were now deployed on disk arrays (rather than a SLED). This "one-file system-per-disk-array" bond presented a dilemma to storage administrators, as one could either use a single file system paired with a large storage array for managing all data together, thereby trading off flexibility in policy specification, or create several small file systems, one per unit of administration, using multiple small storage arrays, thereby sacrificing performance (lesser spindles per file system), and resource utilization (storage space is not shared across arrays). It was soon realized that a separation of file management from storage management was required to eliminate this tradeoff and Volume Managers were born.

Volume managers broke the "one-file system-per-disk-array" bond by exploiting the logical-disk abstraction introduced by RAID and adding yet another layer of indirection—a logical volume [99]. As volume mangers were integrated at the block level, all file systems continued to work unmodified, mapping user files and directories to logical volume blocks. The volume manager, in turn, mapped these logical volume blocks into physical blocks stored on RAID arrays, thereby decoupling data management and storage management. As multiple logical volumes could now be consolidated in shared storage arrays, administrators could use file systems as units of policy enforcement without having to sacrifice performance or resource utilization.

1.1.4 User-Level Metadata Management

The simplicity of the hierarchical organization coupled with the absence of a data model-enforced file format resulted in file systems being adopted as the data store of choice in several application domains. However, as these domains evolved, file systems were forced to offer interfaces for storing and retrieving arbitrary, application-specific, out-of-band metadata in addition to the standard POSIX metadata. As the amount of metadata grew, applications faced two issues borne out of the use of file systems as metadata stores. First, most file systems were not optimized for storing and retrieving extremely large amounts of metadata. Second, file systems were incapable of supporting new domain-specific naming schemes.

In order to solve these two problems while retaining compatibility with existing storage installations, several customized applications (like desktop search) and file format libraries (like HDF5) were developed, thereby adding the third layer to the storage stack—the domain-specific metadata management layer. As components (or protocols) in this layer run in user space outside of the file system, they were no longer limited by the hierarchical naming constraint. Thus, these metadata management tools and applications developed rapidly to include a wide gamut of features ranging from customized, search-friendly layout of metadata, efficient multidimensional indexing, and even domain-specific naming schemes. Thus, file systems were relegated to the role of "dumb" data stores that handle persistence of data created by protocols in the metadata management layer.

1.1.5 Second-Level Flash Integration

Over the past few years, NAND flash-based solid state storage devices have gained wide spread adoption in both personal and enterprise computing. Flash devices have different idiosyncrasies than magnetic disks [7]. First, flash devices are read/written at the granularity of a page—a block of data whose size is usually 4-KB (similar to the block size used by many file systems). Second, once programmed, flash pages need to be erased before they can be reprogrammed, and such erasures can be done only at the granularity of a group of pages referred to as a flash block. Third, flash cells can be erased only a limited number of times after which they wear out and become incapable of storing data reliably.

As flash devices failed to conform to the "traditional block device" stereotype, they could not be integrated into the traditional storage stack. As the age-old LBA bond was no longer applicable to raw flash devices, new file systems and RAID layer protocols were required to map file/block interfaces to the new flash interface. Thus, in order to directly interface with flash devices, one had to sacrifice compatibility with either traditional file systems or RAID layer protocols. While some flash manufacturers took this approach and developed entirely new

storage stacks [48], other industry players opted for a more compatibility-friendly route—adding a new layer to the storage stack below the RAID layer. This new layer, commonly referred to as the Flash Translation Layer [28], is responsible for mapping the traditional LBA interface above to flash-specific interface below. Protocols in this layer are typically implemented by a controller resident in the flash-based Solid State Disk (SSD) and perform several tasks in addition to managing the LBA-physical page mapping like wear-leveling and garbage collection. With this new layer in place, flash-based SSDs could be used like any other block device in combination with traditional file systems and RAID layer protocols.

With compatibility issues out of the way, storage system designers began integrating flash-based SSDs in various capacities. As both cost/GB and read/write latency of flash was ideally positioned between DRAM and HDDs, flash was integrated as a persistent second-level cache (below DRAM) in enterprise servers [21]. As the cost/GB of flash fell, and as the power benefits of flash became more apparent, researchers from industry and academia demonstrated the benefits of replacing DRAM with large amounts of flash-based SSDs. This, in concert with a marked increase in SSD densities due to advances in manufacturing practices, encouraged designers to integrate flash as a storage tier alongside HDDs for storing primary data rather than caching it [42], [51], [37].

As modern data centers switched to virtualization-based server consolidation, primary data storage also became consolidated, often in the form of a shared Network Attached Storage (NAS) appliance. With such a shift, it soon became apparent that the use of flash as a second-level cache on the client side would provide several benefits compared to server integration [18]. However, a client-side flash cache would have to be integrated at the hypervisor level in order to support snapshoting and caching of blocks belonging to virtual machine disk images. Thus, following suit with RAID and Volume Manager integration, caching and tiering algorithms were also integrated at the block level using the LBA abstraction, thus, simplifying hypervisor integration in virtualized data centers, and retaining backwards compatibility with existing file systems in nonvirtualized enterprise installations.

1.2 Retiring the Traditional Stack

The traditional stack shown in Figure 1.1 adopts one of many possible ways of layering storage protocols. As we mentioned earlier, this current layering was born out of an overarching emphasis on compatibility, in contrast to the network stack, for instance, where the division of labor between layers was the product of a modular design. The result of such compatibility-driven protocol layering is a storage stack riddled with design assumptions that have been invalidated time and again by the addition of new layers or integration of new storage hardware.

For instance, the "one-file system-per-disk" bond based on which file systems performed several optimizations was invalidated by the addition of the RAID layer. Similarly, several file system optimizations that aimed at avoiding random I/O and improving sequential accesses were invalidated when flash-based storage was integrated into the stack.

These invalidated design assumptions cause several issues that impact performance, reliability, and flexibility of the traditional stack. For instance, it has been shown that a few widely-used RAID array configurations can adversely interact with certain file system layout optimizations causing several performance issues [91]. Similarly, the block-level integration of volume managers necessitated cross-layer changes for adding new features like online expansion and shrinking of logical volumes. This, in turn, complicated storage administration as a simple task of adding a new storage device was transformed into an error-prone multi-stage operation which involved manually requesting each layer to update its internal data structures. Similarly, the block-level integration of caching algorithms impacts reliability as it complicates consistency management and crash recovery due to reordering of erstwhile-ordered file system operations. Given these issues, and several others which we will present later in this thesis, we believe that the traditional stack must be retired in favor of a fresh, ground-up redesign.

1.3 Introducing Loris

In this thesis, we present the design and implementation of Loris, a storage stack that solves all performance, flexibility, reliability, and heterogeneity issues of the traditional stack by design. In doing so, we present a new, clean-slate layering of storage protocols that sacrifices backwards compatibility in favor of modularity.

Figure 1.2 shows the two steps that mark the conceptual transition from the traditional stack to the Loris stack. The first conceptual step is the decomposition of the file system layer into the Naming, Cache, and Layout sublayers. Protocols in each sublayer can be seen as providing naming scheme implementations, first-level cache management, and device-specific layout management respectively. Since these protocols were originally implemented by the file system, they work with files, in contrast to the RAID layer protocols, that operate on semantically-unrelated data blocks. The second conceptual step involves promoting the device agnostic protocols, namely, those belonging to the traditional RAID layer (RAID, volume management, caching, etc.), to the file level by repositioning them above the device-specific Layout sublayer. We refer to this new layering of storage protocols as the Loris storage stack.

The rationale behind these two transitions is quite straightforward. The first transition improves modularity by identifying individual protocols and separating them. The second step essentially groups protocols into two categories, namely,

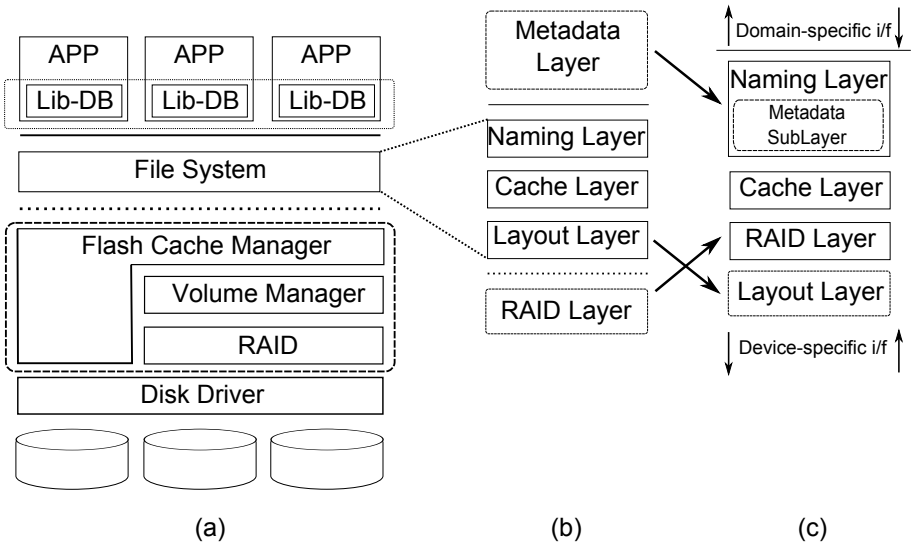


Figure 1.2: The figure shows the conceptual transition from the traditional stack (a) to the Loris stack (c).

device specific and device agnostic, and layers all device-agnostic protocols at the file level. As a result, all layers in the Loris stack are file aware, and all RAID layer protocols in Loris, be it RAID, volume management or flash caching, operate on files rather than blocks. As we will show later in this thesis, Loris uses a file-based indirection, in contrast to a block-based logical disk indirection, for implementing various features like file volume virtualization, thin provision, per-file snapshoting and even open-close versioning.

Loris supports a novel storage model, which we refer to as File Pooling, for managing (adding, removing, hot swapping) devices online. Administrative operations, like addition or removal of storage devices, that typically involve multiple error-prone steps in the traditional stack can be accomplished using a single step in Loris because of File Pooling.

Loris is capable of supporting several naming layers due to its modular division of labor. Thus, Loris can act as a framework for implementing domain-specific metadata management systems. In fact, in addition to the standard POSIX-based naming scheme, Loris also supports a new naming layer protocol that provides highly-optimized LSM-tree-based metadata storage, real- time inline indexing of user-specified attributes, and an attribute-based query language for searching and organizing metadata.

As we mentioned earlier, Loris also integrates second-level flash management algorithms at the file level. However, despite its file level integration, Loris has been designed to support efficient, fine-grained subfile tiering or caching of “hot”

data in the flash cache. Unlike the traditional stack, where both the file system and block layers have to implement redundant consistency management for providing crash-consistent flash caching, Loris uses a single, unified scheme for protecting cross-stack metadata from power failures and OS crashes.

1.4 Contributions of this Thesis

Now that we have provided a preview of the problems that plague the traditional stack and outlined the advantages of the new conceptual layering of storage protocols in Loris, we will now explicitly list the contributions of this thesis.

1. A new storage stack
    - The design and implementation of the Loris stack with a new layering of storage protocols
2. File-level RAID and volume management
    - File Pools, a novel storage model that simplifies storage administration
    - A single, unified, Loris-based virtualization infrastructure that supports file volume virtualization, snapshoting, per-file versioning
3. Efficient metadata management
    - A Loris-based, modular metadata management system that provides 1) LSM-tree-based metadata storage, 2) real-time indexing infrastructure that uses LSM-trees for maintaining attribute indices, and 3) scalable metadata querying using an attribute-based query language
4. File-level flash integration
    - A Loris-based framework that enables development of file-level, hybrid, primary storage systems that use SSDs in concert with HDDs in various capacities, and an empirical study of the pros and cons of various caching and tiering algorithms.
    - The design and implementation of a Loris-based, host-side flash cache that performs fine-grained, subfile caching without any consistency issues

We would like to explicitly mention here that this thesis does not cover techniques that protect the Loris stack from crashes due to power failures or operating system bugs. These topics, together with other reliability extensions, form a part of another thesis (by David van Moolenbroek).

1.5 Organization of this Thesis

This thesis is organized as a collection of self-contained, refereed publications, each focusing on one of the aforementioned contributions.

- Chapter 2 focuses on RAID algorithms. We first describe in detail various issues that plague the block-level integration of RAID algorithms in the traditional stack. Then, we present Loris, our new storage stack, and show how the file-level integration of RAID algorithms in Loris solves all these issues by design. To this end, we provide a detailed description of the data structures and algorithms used by the four Loris layers for implementing reliable, flexible, heterogeneity-friendly, file-level RAID. Chapter 2 appeared in *Proceedings of the Pacific Rim International Symposium on Dependable Computing (PRDC'10)* [9]. We would like to explicitly mention here that this work was done in collaboration with David van Moolenbroek. Thus, the material presented here will also appear in David van Moolenbroek's thesis.

- Chapter 3 focuses on device management and file volume virtualization. After describing the flexibility and heterogeneity issues with traditional volume managers, we present File Pools, our Loris-based storage model. In addition, we also present the design extensions to Loris for supporting file volume virtualization, and show how the modular division of labor between various Loris layers unifies the implementation of volume snapshoting and per-file open/close versioning. Chapter 3 appeared in *Proceedings of the 27th Symposium on Mass Storage Systems and Technologies(MSST'11)* [10].

- Chapter 4 focuses on scalable metadata management. After providing a detailed overview of various problems that affect performance and scalability of user-level metadata management systems, we show how Loris can act as a framework for supporting various domain-specific metadata solutions. In doing so, we present the design and implementation of our new Loris naming layer that uses LSM-trees for high-performance metadata lookup/storage and real-time indexing of user-specified attributes. We also show how an attribute-based, domain-specific query language can be used side-by-side with a traditional POSIX-based naming layer to provide search-based access to data stored in a hierarchical file system. Chapter 3 appeared in *Proceedings of the Sixth International Conference on Networking, Architecture, and Storage (NAS'11)* [101].

- Chapter 5 focuses on the design tradeoffs involved in building primary data storage systems, also known as hybrid storage systems, that use high-performance flash in concert with high-density, low-cost HDDs for improving price and performance. We first show how Loris can be used as a framework for implementing file-level hybrid storage systems and describe its

advantages over other block-level systems. Using Loris as a framework, we then present the design, implementation, and evaluation of several tiering and caching policies that use flash-based SSDs in various capacities. In doing so, we identify tradeoffs inherent to each approach and discuss the ramifications of these tradeoffs on the design of future hybrid storage systems. Chapter 5 appeared in *Proceedings of the 28th Symposium on Mass Storage Systems and Technologies (MSST'12)* [11].

- Chapter 6 focuses on host-side flash caching in data centers. After describing various consistency issues that affect block-level integration of flash caching algorithms, we present the design and implementation of our Loris-based flash caching system that can perform fine-grained, block-granular buffering of "hot" data in the flash cache. We also present a comparative evaluation to illustrate the benefit of file-level flash cache management over the block-level approach. Chapter 6 appeared in *Proceedings of the 19th International Conference on Parallel and Distributed Systems (ICPADS'13)* [12].

Chapter 2

Loris - A Dependable, Modular, File-Based Storage Stack

Abstract

The arrangement of file systems and volume management/RAID systems, together commonly referred to as the storage stack, has remained the same for several decades, despite significant changes in hardware, software and usage scenarios. In this paper, we evaluate the traditional storage stack along three dimensions: reliability, heterogeneity and flexibility. We highlight several major problems with the traditional stack. We then present Loris, our redesign of the storage stack, and we evaluate several reliability, availability and performance aspects of Loris.

2.1 Introduction

Over the past several years, the storage hardware landscape has changed dramatically. A significant drop in the cost per gigabyte of disk drives has made techniques that require a full disk scan, like *fsck* or whole disk backup, prohibitively expensive. Large scale installations handling petabytes of data are very common today, and devising techniques to simplify management has become a key priority in both academia and industry. Research has revealed great insights into the reliability characteristics of modern disk drives. "Fail-partial" failure modes [76] have been studied extensively and end-to-end integrity assurance is more important now than ever before. The introduction of SSDs and other flash devices is sure to bring about a sea change in the storage subsystem. Radically different price/performance/reliability trade-offs have necessitated rethinking several aspects of file and storage management [48; 28]. In short, the storage world is rapidly changing and our approach to making storage dependable must change, too.

Traditional file systems were written and optimized for block-oriented hard disk drives. With the advent of RAID techniques [73], storage vendors started developing high-performance, high-capacity RAID systems in both hardware and software. The block-level interface between the file system and disk drives provided a convenient, backward-compatible abstraction for integrating RAID algorithms.

As installations grew in size, administrators needed more flexibility in managing file systems and disk drives. Volume managers [99] were designed as block-level drivers to break the "one file system per disk" bond. By providing logical volumes, they abstracted out details of physical storage and thereby made it possible to resize file systems/volumes on the fly. Logical volumes also served as units of policy assignment and quota enforcement. Together, we refer to the RAID and volume management solutions as the *RAID layer* in this paper.

This arrangement of file system and RAID layers, as shown in Figure 2.1(a), has been referred to as the storage stack [24]. Despite several changes in hardware landscape, the traditional storage stack has remained the same for several decades. In this paper, we examine the block-level integration of RAID and volume management along three dimensions: reliability, flexibility, and heterogeneity. We highlight several major problems with the traditional stack along all three dimensions. We then present Loris, our new storage stack. Loris improves modularity by decomposing the traditional file system layer into several self-contained layers, as shown in Figure 2.1(b). It improves reliability by integrating RAID algorithms at a different layer compared to the traditional stack. It supports heterogeneity by providing a file-based stack in which the interface between layers is a standardized file interface. It improves flexibility by automating device administration and enabling per-file policy selection.

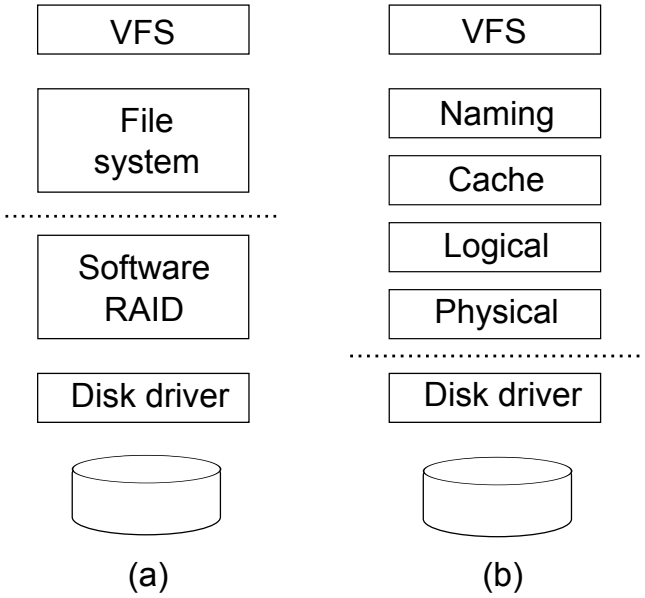


Figure 2.1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file-aware, the layers below are not.

This paper is structured as follows. In Sec. 2, we explain in detail the problems associated with the traditional stack. In Sec. 3, we briefly outline some attempts taken by others in redesigning the storage stack and also explain why other approaches fail to solve all the problems. In Sec. 4, we introduce Loris and explain the responsibilities and abstraction boundaries of each layer in the new stack. We also sketch the realization of these layers in our prototype implementation. In Sec. 5, we present the advantages of Loris. We then evaluate both performance and reliability aspects of Loris in Sec. 6 and conclude in Sec. 7.

2.2 Problems with the Traditional Storage Stack

In this section, we present some of the most important problems associated with the traditional storage stack. We present the problems along three dimensions: reliability, flexibility and heterogeneity.

2.2.1 Reliability

The first dimension is reliability. This includes the aspects of data corruption, system failure, and device failure.

Data Corruption

Modern disk drives are not "fail-stop." Recent research has analyzed several "fail-partial" failure modes of disk drives. For example, a *lost write* happens when a disk does not write a data block. A *misdirected* write by the disk results in a data block being written at a different position than its intended location. A *torn write* happens when a write operation fails to write all the sectors of a multisector data block. In all these cases, if the disk controller (incorrectly) reports back success, data is silently corrupted. This is a serious threat to data integrity and significantly affects the reliability of the storage stack.

File systems and block-level storage systems detect data corruption by employing various checksumming techniques [86]. The level of reliability offered by a checksumming scheme depends heavily on what is checksummed and where the checksum is stored. Checksums can be computed on a per-sector or per-block (file system block) basis, where a block is typically 2, 4, 8, or more sectors. Using per-sector checksums does not protect one against any of the failure cases mentioned above if checksums are stored with the data itself. Block checksums, on the other hand, protect against torn writes but not against misdirected or lost writes.

In yet another type of checksumming, called *parental checksumming*, the checksum of a data block is stored with its parent. For instance, a parental checksumming implementation could store block checksums in the inode, right next to the block pointers. These checksums would then be read in with the inode and used for verification. Formal analysis has verified that parental checksumming detects all of the aforementioned sources of corruption [55].

However, using parental checksumming in the current storage stack increases the chance of data loss significantly [55]. Since the parental relationship between blocks is known only to the file system, parental checksums can be used only by the file system layer. Thus, while file system initiated reads can be verified, any reads initiated by the RAID layer (for partial writes, scrubbing, or recovery) escape verification. As a result, a corrupt data block will not only go undetected by the RAID layer, but could also be used for parity recomputation, causing parity pollution [55], and hence data loss.

System Failure

Crashes and power failures pose a *metadata consistency* problem for file systems. Several techniques like soft updates and journaling have been used to reliably update metadata in the face of such events. RAID algorithms also suffer from a *consistent update* problem. Since RAID algorithms write data to multiple disk drives, they must ensure that the data on different devices are updated in a consistent manner.

Most hardware RAID implementations use NVRAM to buffer writes until they

are made durable, cleanly side-stepping this problem. Several software RAID solutions, on the other hand, resort to whole disk *resynchronization* after an unclean shutdown. During resynchronization, all data blocks are read in, parity is computed, and the computed parity is verified with the on-disk parity. If a mismatch is detected, the newly computed parity is written out replacing the on-disk parity.

This approach—in addition to becoming increasingly impractical due to the rapid increase in disk capacity—has two major problems: (1) it increases the vulnerability window within which a second failure can result in data loss. This problem is also known as the "RAID write hole", and (2) it adversely affects availability, as the whole disk array has to be offline during the resynchronization period [16].

The other approach adopted by some software RAID implementations is journaling a block bitmap to identify regions of activity during the failure. While this approach reduces resynchronization time, it has a negative impact on performance [25], and results in functionality being duplicated across multiple layers.

Device Failure

Storage array implementations protect against a fixed number of disk failures. For instance, a RAID 5 implementation protects against a single disk failure. When an unexpected number of failures occur, the storage array comes to a grinding halt. An ideal storage array however, should degrade gracefully. The amount of data inaccessible should be proportional to the number of failures in the system. Research has shown that to achieve such a property, a RAID implementation must provide: (1) selective replication of metadata to make sure that the directory hierarchy remains navigable at all times, and (2) fault-isolated positioning of files so that a failure of any single disk results in only files on that disk being inaccessible [88].

By recovering *files* rather than *blocks*, file-level RAID algorithms reduce the amount of data that must be recovered, thus shrinking the vulnerability window before a second failure. Even a second failure during recovery results in the loss of only some file(s), which can be restored from backup sources, compared to block-level RAID where the entire array must be restored. None of these functionalities can be provided by the traditional storage stack as the traditional RAID implementation operates strictly below a block interface.

2.2.2 Flexibility

We discuss two points pertaining to flexibility: management and policy assignment.

Management Flexibility

While traditional volume managers make device management and space allocation more flexible, they introduce a series of complex, error prone administrative operations, most of which should be automated. For instance, a simple task such as adding a new disk to the system involves several steps like creating a physical volume, adding it to a volume group, expanding logical volumes and finally resizing file systems. While new models of administering devices, like the storage pool model [6], are a huge improvement, they still suffer from other problems, which we will describe in the next section.

In addition to device management complexity, software-based RAID solutions expose a set of tunable parameters for configuring a storage array based on the expected workload. It has been shown that an improperly configured array can render layout optimizations employed by a file system futile [91]. This is an example of the more general "information gap" problem [24]— different layers performing different optimizations unaware of the effect they might have on the overall performance.

Policy Assignment Flexibility

Different files have different levels of importance and need different levels of protection. However, policies like the RAID level to use, encryption, and compression, are only available on a per-volume basis rather than on a per-file basis. Several RAID implementations even lock-in the RAID levels and make it impossible to migrate data between RAID levels. In cases where migration is supported, it usually comes at the expense of having to perform a full-array dump and restore. In our view, an ideal storage system should be flexible enough to support policy assignment on a per-file, per-directory, or a per-volume basis. It should support migration of data between RAID levels on-the-fly without affecting data availability.

2.2.3 Heterogeneity Issues

New devices are emerging with different data access granularities and storage interfaces. Integrating these devices into the storage stack has been done using two approaches that involve either extending either the file system layer, or the block-level RAID layer with new functionality.

The first approach involves building file systems that are aware of device-specific abstractions [48]. However, as the traditional block-based RAID layer exposes a block interface, it is incompatible with these file systems. As a result, RAID and volume management functionalities must be implemented from scratch for each device family.

The second approach is to be backward compatible with traditional file systems. This is typically done by adding a new layer that translates block requests from the file system to device-specific abstractions [28]. This layer cannot be integrated between the file system and RAID layers, as it is incompatible with the RAID layer. Hence, such a layer has to either reimplement RAID algorithms for the new device family, or be integrated below the RAID layer. This integration retains compatibility with both traditional file system and RAID implementations. However, this has the serious effect of widening the information gap by duplicating functionality. For instance, both the file system and translation layers now try to employ layout optimizations – something that is completely unwarranted. The only way to avoid this duplication is by completely redesigning the storage stack from scratch.

2.3 Solutions Proposed in the Literature

Several approaches have been taken to solving some of the problems mentioned in the previous section. However, none of these approaches solve all these problems by design. In this section, we highlight only the most important techniques. We classify the approaches taken into four types, namely: (1) inferring information, (2) sharing information, (3) refactoring the storage stack, and (4) extending the stack with stackable filing.

One could compensate for the lack of information in the RAID layer by having it infer information about the file system layer. For instance, semantically smart disks [88] infer file system specific information (block typing and structure information), and use the semantic knowledge to improve RAID flexibility, availability, and performance. However, by their very nature, they are file-system-specific, making them nonportable.

Instead of inferring information, one could redesign the interface between the file system and RAID layers to share information. For example, ExRAID [24], a software RAID implementation, provides array related information (such as disk boundaries and transient performance characteristics of each device) to an informed file system (I-LFS), which uses it to make informed data placement decisions.

While both inferring and sharing information can be used to add new functionality, they do not change the fundamental division of labor between layers. Hence, most of the problems we mentioned remain unresolved.

A few projects have refactored the traditional storage stack. For instance, ZFS [6]'s storage stack consists of three layers, the ZFS Posix Layer (ZPL), the Data Management Unit (DMU), and the Storage Pool Allocator (SPA). ZFS solves the reliability and flexiblity problems we mentioned earlier by merging block allocation with RAID algorithms in its SPA layer. SPA exposes a virtual block

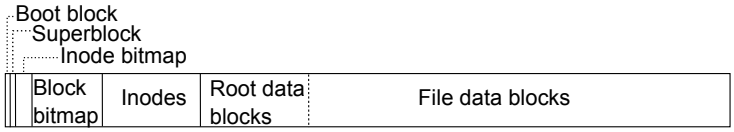


Figure 2.2: High-level overview of the on-disk layout used by the physical layer prototype.

abstraction to DMU and acts as a multidisk block allocator. However, because SPA exposes a block interface, it suffers from the same heterogeneity problems as the RAID layer. In addition, we believe that layout management and RAID are two distinct functionalities that should be modularized in separate layers.

RAIF [49] provides RAID algorithms as a stackable file system. RAID algorithms in RAIF work on a per-file basis. As it is stackable, it is very modular and can be layered on any file system, making it device independent. While this does solve the flexibility and heterogeneity problems, it does not solve the reliability problems.

2.4 The Design of Loris

We now present our new storage stack, Loris. The Loris stack consists of four layers in between the VFS layer and the disk driver layer, as shown in Figure 2.1(b). Within the stack, the primary abstraction is the *file*. Each layer offers an interface for creating and manipulating files to the layer above it, exposing per-file operations such as *create*, *read*, *write*, *truncate*, and *delete*. A generic *attribute* abstraction is used to both maintain per-file metadata, and exchange information within the stack. The *getattr* and *setattr* operations retrieve and set attributes. Files and attributes are stored by the lowest layer.

We implemented a Loris prototype on the MINIX 3 multiserver operating system [44]. The modular and fault-tolerant structure of MINIX 3 allows us to quickly and easily experiment with invasive changes. Moreover, we plan to apply ongoing research in the areas of live updates [34] and many-core support to our work. MINIX 3 already provides VFS and the disk drivers, each running as a separate process, which improves dependability by allowing failed OS components to be replaced on the fly.

The four layer implementations of the prototype can be combined into one process, or separated out into individual processes. The single-process setup allows for higher performance due to fewer context switches and less memory copying overhead. The multiprocess setup by nature imposes a very strict separation between layers, provides better process fault isolation in line with MINIX 3's dependability objectives [44], and is more live-update-friendly [34]. The difference

between these configurations is abstracted away by a common library that provides primitives for communication and data exchange.

We will now discuss each of the layers in turn, starting from the lowest layer.

2.4.1 The Physical Layer

The lowest layer in the Loris stack is the *physical layer*. The physical layer algorithms provide device-specific layout schemes to store files and attributes. They offer a *fail-stop physical file* abstraction to the layers above it. By working on physical files, the rest of the layers are isolated from device-specific characteristics of the underlying devices (such as access granularity).

As the physical files are fail-stop, every call that requests file data or attributes, returns either a result that has been verified to be free of corruption or an error. To this end, every physical layer algorithm is required to implement parental checksumming. To repeat, above the physical layer, there is no silent corruption of data. A torn, lost, or misdirected write is converted into a hard failure that is passed upward in the stack.

In our prototype, each physical device is managed by a separate, independent instance of one of the physical layer algorithms. We call such an instance a *module*. Each physical module has a global *module identifier*, which it uses to register to the logical layer at startup.

On-Disk Layout

The on-disk layout of our prototype is based directly on the MINIX 3 File System (MFS) [97], a traditional UNIX-like file system. We have extended it to support parental checksums. Figure 2.2 shows a high-level view of the layout. We deliberately chose to stay close to the original MFS implementation so that we could measure the overhead incurred by parental checksumming.

Each physical file is represented on disk by an *inode*. Each inode has an *inode number* that identifies the physical file. The inode contains space to store persistent attributes, as well as 7 direct, one single indirect and one double indirect *safe block pointers*. Each safe block pointer contains a block number and a CRC32 checksum of the block it points to. The single and double indirect blocks store such pointers as well, to data blocks and single indirect blocks, respectively. All file data are therefore protected directly or indirectly by checksums in the file inode.

The inodes and other metadata are protected by means of three special on-disk files. These are the inode bitmap file, the block bitmap file, and the "root file." The bitmap file inodes and their indirect blocks contain safe block pointers to the bitmap blocks. The root file forms the hierarchical parent of all the inodes—its data blocks contain checksums over all inodes, including the bitmap file inodes.

The checksums in the root file are stored and computed on a per-inode basis, rather than a per-block basis. The checksum of the root file's inode is stored in the superblock.

Figure 2.3 shows the resulting parental checksumming hierarchy. The superblock and root data blocks contain only checksums; the inodes and indirect blocks contain safe block pointers. Please note that this hierarchy is used only for checksumming—unlike copy-on-write layout schemes [45], blocks are updated in-place.

Delayed Checksum Verification

One of the main drawbacks of parental checksumming is cascading writes. Due to the inherent nature of parental checksumming, a single update to a block could result in several blocks being updated all the way up the parental tree to the root. Updating all checksums in the mainstream write path would slow down performance significantly.

The physical layer prevents this by delaying the checksum computation, using a small metadata block cache that is type-aware. By knowing the type of each block, this cache makes it possible to perform checksum computation only when a block is written to or read from the underlying device. For instance, by knowing whether a block is a bitmap or inode block, it can compute and update checksums in the bitmap and root file inodes lazily, that is, only right before flushing these metadata blocks from the cache.

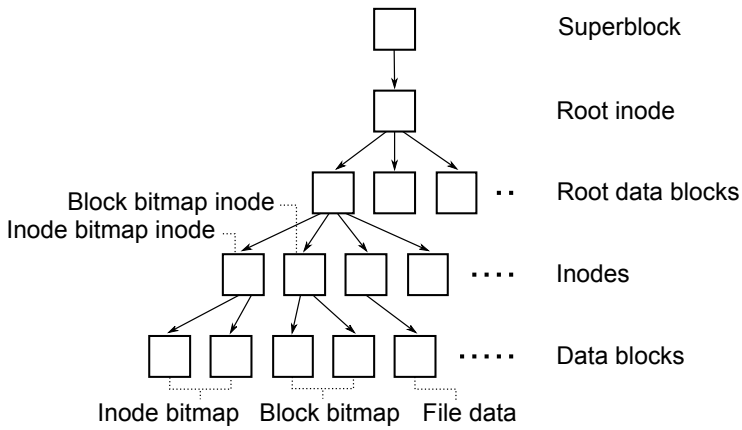


Figure 2.3: Parental checksumming hierarchy used by the physical layer prototype. With respect to parental checksumming, the two special bitmap files are treated as any other files. Indirect blocks have been omitted in this figure.

Error Handling

Parental checksumming allows the physical layer to detect corrupt data. When a physical module detects a checksum error in the data or indirect block of a file, it marks that portion of the file as corrupted. If the file's inode checksum, as stored in the root file, does not match the checksum computed over the actual inode, then the entire file is marked as corrupted. In both cases, reads from the corrupt portion will result in a checksum error being returned to the logical layer. The physical module uses two attributes in the file's inode, begin range and end range, to remember this *sick range*.

For instance, consider a read request to a physical module for the first data block of a file. If the physical module detects a checksum mismatch on reading in the data block, it sets the begin and end range attributes to 0 and 4095 respectively, and responds back to the logical layer with a checksum error. We will detail the recovery process initiated by the logical layer when we describe its error handling mechanism.

In contrast to inodes and data blocks, if a part of the other on-disk metadata structures is found corrupted, the physical module will shut down for offline repair. While we could have supported metadata replication to improve reliability, we chose not to do so for the first prototype to stay as close as possible to the original MFS implementation in order to get honest measurements.

If the underlying device driver returns an error or times out, the physical module will retry the operation a number of times. Upon repeated failure, it returns back an I/O error to the logical layer. An I/O error from the physical module is an indication to the logical layer that a fatal failure has occurred and that the erroneous device should not be used anymore.

2.4.2 The Logical Layer

The logical layer implements RAID algorithms on a per-file basis to provide various levels of redundancy. The logical layer offers a *reliable logical file* abstraction to the layers above. It masks errors from the physical layer whenever there is enough data redundancy to recover from them. One logical file may be made up of several independent physical files, typically each on a different physical module, and possibly at different locations. As such, the logical layer acts as a centralized multiplexer over the individual modules in the physical layer.

Our prototype implements the equivalents of RAID levels 0, 1, and 4—all of these operate on a per-file basis. There is only one module instance of the logical layer, which operates across all physical modules.

File Mapping

The central data structure in our logical layer prototype is the *mapping*. The mapping contains an entry for every logical file, translating *logical file identifiers* to *logical configurations*. The logical configuration of a file consists of (1) the file's RAID level, (2) the stripe size used for the file (if applicable), and (3) a set of one or more physical files that make up this logical file, each specified as a physical module and inode number pair. The RAID level implementations decide how the physical files are used to make up the logical file.

The *create* operation creates a mapping entry for a given logical file identifier, with a given configuration (more about this later). For all other file operations coming in from above, the logical layer first looks up the file's logical configuration in the mapping. The corresponding RAID algorithm is then responsible for calling appropriate operations on physical modules.

Figure 2.4 shows how a logical file that is striped across two devices, is constructed out of two independent physical files. The mapping entry for this file F1 could look like this: F1=<raidlevel=0, stripesize=4096, physicalfiles=<D1:I1, D2:I2>>. The entry specifies a RAID level of 0, a stripe size of 4096 bytes, and two physical files: file I1 on physical module D1 and file I2 on physical module D2. Now consider a read request for the first 16384 bytes of this file coming down to the logical layer. Upon receiving the read request, the logical layer looks up the entry for F1 in its mapping, and passes on the call to the RAID 0 algorithm. The RAID 0 code uses the entry to determine that logical bytes 0-4095 are stored as bytes 0-4095 in physical file D1:I1, logical bytes 4096-8191 are stored as bytes 0-4095 in file D2:I2, logical bytes 8192-12287 are stored as bytes 4096-8191 in file D1:I1, and logical bytes 12288-16383 are stored as bytes 4096-8191 in file D2:I2.

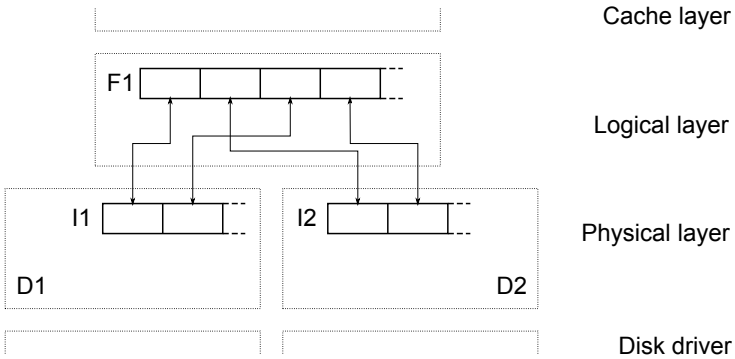


Figure 2.4: An example of the file abstractions provided by the logical and physical layers. The logical layer exposes a logical file, F1, which is constructed out of two physical files, namely I1 on physical module D1 and I2 on physical module D2, by means of striping.

The logical layer issues two read requests, one to D1 for the first 8192 bytes of I1, and the other to D2 for the first 8192 bytes of I2. The results are combined to form a "flat" result for the layer above.

The mapping itself is a logical file. The logical configuration of this file is hardcoded. The mapping file is crucial for accessing any other files, and is therefore mirrored across all physical modules for increased dependability. It uses the same static inode number on all physical modules.

Error Handling

When the logical layer gets a checksum error in response to an operation on a physical module, it will restore the correct data for the file involved in the operation if enough data redundancy is present. If on-the-fly restoration is not possible, the logical layer fails the operation with an I/O error.

For instance, let us consider a read request for the first block of a file mirrored on two physical modules P1 and P2. If P1 responds back with a checksum error, the logical layer first retrieves the begin and end sick range attributes from P1. Assuming that only the first block was corrupt, these values would be 0 and 4095. The logical layer then issues a read request to module P2, for data in the range 0–4095. If this read request succeeds, the logical layer issues a write request to P1 for this data range, thereby performing on-the-fly recovery. It finally clears the sick range by resetting begin and end ranges to their defaults.

When the logical layer gets an I/O error from a physical module, it considers this to be a permanent error, and disables the physical module. The logical layer will continue to serve requests for affected files from redundant copies where possible, and return I/O errors otherwise.

2.4.3 The Cache Layer

The cache layer caches file data in main memory. This layer may be omitted in operating systems that provide a unified page cache. As MINIX 3 does not have a unified page cache, our prototype implements this layer. The cache layer is also needed on systems that do not have any local physical storage such as PDAs and other small mobile devices.

File-Based Caching

The prototype cache is file-aware and performs readahead and eviction of file data on a per-file basis. Files are read from the lower layers in large readahead chunks at a time, and only entire files are evicted from the cache. The cache maintains file data at the granularity of memory pages.

Early experiments showed a large performance penalty incurred by small file writes. Small file writes were absorbed completely by the cache until a sync re-

quest was received or the cache needed to free up pages for new requests. During eviction, each file would be written out by means of an individual *write* operation, forcing the physical layer to perform a large number of small random writes. To counter this problem, we introduced a *vwrite* call to supply a vector of write operations for several files. The cache uses this call to pass down as many dirty pages as possible at once, eventually allowing the physical modules to reorder and combine the small writes.

Problems with Delayed Allocation

The cache delays writes, so that *write* calls from above can be satisfied very quickly. This results in allocation of data blocks for these writes to be delayed until the moment that these blocks are flushed out. This delayed allocation poses a problem in the stack. Because of the abstraction provided by the logical layer, the cache has no knowledge about the devices used to store a file, nor about the free space available on those devices. Therefore, when a write operation comes in, the cache cannot determine whether the write will eventually succeed.

Although we have not yet implemented a solution for this in our prototype, the problem can be solved by means of a free-space reservation system exposed by the physical modules through the logical layer to the cache.

2.4.4 The Naming Layer

The naming layer is responsible for naming and organizing files. Different naming layer algorithms can implement different naming schemes: for example, a traditional POSIX style naming scheme, or a search-oriented naming scheme based on attributes.

POSIX Support

Our prototype implements a traditional POSIX naming scheme. It processes calls coming from the VFS layer above, converting POSIX operations into file operations.

Only the naming layer knows about the concept of directories. Below the naming layer, directories are stored as files. The naming layer itself treats directories as flat arrays of statically sized entries, one per file. Each entry is made up of a file name and a logical file identifier. Again, this layout was chosen for simplicity and to stay close to the original MFS implementation for comparison purposes. A new naming module could implement more advanced directory indexing.

The naming layer is also responsible for maintaining the POSIX attributes of files (file size, file mode, link count, and so on). It uses Loris attributes for this: it uses the *setattribute* call to send down POSIX attribute changes, which are ultimately processed and stored by the physical layer in the file's inode.

Policy Assignment

When creating a new file, the naming layer is responsible for picking a new logical file identifier, and an initial logical configuration for the file. The logical configuration may be picked based on any information available to the naming layer: the new file's name, its containing directory, its file type, and possibly any flags passed in by the application creating the file. The chosen logical configuration is passed to lower layers in the *create* call in the form of attributes.

By default, directories are mirrored across all devices in order to provide graceful degradation. Upon getting a *create directory* request from VFS, the naming layer picks a new file identifier for the directory, and sends down a *create* call to the cache, with RAID level 1 specified as the file's logical policy. The cache forwards the call to the logical layer. The logical layer creates a new entry in the mapping for this file, and forwards the create call to all of the physical modules. Upon return, the logical layer stores the resulting inode numbers in the mapping entry as well.

2.5 The Advantages of Loris

In this section, we highlight how Loris solves all the problems mentioned in Sec. 2 by design. This section has been structured to mirror the structure of Sec. 2 so that readers can match the problems with their corresponding solutions one-to-one.

2.5.1 Reliability

We now explain how Loris protects against the three threats to data integrity.

Data Corruption

As RAID algorithms are positioned in the logical layer, all requests, both user application initiated reads and RAID initiated reads, are serviced by the physical layer. Thus, any data verification scheme needs to be implemented only once, in the physical layer, for all types of requests. In addition, since the physical layer is file-aware, parental checksumming can be used for detecting all possible sources of corruption. Thus, by requiring every physical layer algorithm to implement parental checksumming, fail-partial failures are converted into fail-stop failures. RAID algorithms can safely provide protection against fail-stop failures without propagating corruption.

System Failure

Journaling has been used by several file systems to provide atomic update of system metadata [39]. Modified journaling implementations have also been used

to maintain the consistency between data blocks and checksums in the face of crashes [90]. While any such traditional crash recovery techniques can be used with Loris to maintain metadata consistency, we are working on a new technique called *metadata replay*. It protects the system from both hard crashes (power failures, kernel panic, etc.) and soft crashes (modules failing due to bugs). We will provide a brief description of this technique now, but it should be noted that independent of the technique used, Loris does not require expensive whole disk synchronization due to its file-aware nature.

Metadata replay is based on the counterintuitive idea that user data (file data and POSIX attributes), if updated atomically, can be used to regenerate system metadata. To implement this, we have two requirements: (1) a lightweight mechanism to log user data, and (2) some way to restore back the latest consistent snapshot. With highly flexible policy selection in place, the user could log only important files, reducing the overhead of data logging. We plan to add support for selective logging and metadata snapshoting. When a crash occurs, the logical layer coordinates the rollback of all physical layers to a globally consistent state. Then, the logged user data are replayed, regenerating metadata in both logical and physical layers, and bringing the system to new consistent state.

Device Failure

Graceful degradation is a natural extension of our design. Since RAID policies can be selected on a per-file basis, directories can be replicated on all devices while file data need not be, thereby providing selective metadata replication. Since the logical layer is file-aware, fault-isolated placement of files can also be provided on a per-file basis. Furthermore, recovery to a hot spare on a disk failure is faster than a traditional RAID array since the logical layer recovers files. As mentioned earlier, file-granular recovery restores only "live data" by nature, i.e., unused data blocks in all physical layers do not have to be restored. Because traditional RAID operates at the block level, it is unaware of which data blocks hold file data, and has to restore all data blocks in the array.

2.5.2 Flexibility

The new file-oriented storage stack is more flexible than the traditional stack in several ways. Loris supports automating several administrative tasks, simplifies device management, and supports policy assignment at the granularity of files.

Management Flexiblity

Loris simplifies administration by providing a simple model for both device and quota management. It supports automating most of the traditional administrative chores. For instance, when a new device is added to an existing installation, Loris

automatically assigns the device a new identifier. A new physical module corresponding to this device type is started automatically and this module registers itself with the logical layer as a potential source of physical files. From here on, the logical layer is free to direct new file creation requests to the new module. It can also change the RAID policy of existing files on-the-fly or in the background. Thus, Loris supports a pooled storage model similar to ZFS.

File systems in ZFS [6] serve as units of quota enforcement. By decoupling volume management from device management, these systems make it possible for multiple volumes to share the same storage space. We are working on a file volume management implementation for Loris. We plan to modify the logical layer to add support for such a volume manager. File volumes in Loris will be able to provide functionalities similar to ZFS since files belonging to any volume can be allocated from any physical module.

Policy Assignment Flexiblity

Loris provides a clean split between policy and mechanism. For instance, while RAID algorithms are implemented in the logical layer, the policy that assigns RAID levels to files can be present in any layer. Thus, while the naming layer can assign RAID levels on a per-file, per-directory or even per-type basis [49], the logical layer could assign policies on a per-volume basis or even globally across all files.

2.5.3 Heterogeneity

All of the aforementioned functionalities are device-independent. By having the physical layer provide a physical file abstraction, algorithms above the physical layer are isolated from device-specific details. Thus, Loris can be used to set up an installation where disk drives coexist with byte-granular flash devices and Object-based Storage Devices (OSDs), and the administrator would use the same management primitives across these device types. In addition, as the physical layer works directly on the device without being virtualized, device-specific layout optimizations can be employed without creating an information gap.

2.6 Evaluation

In this section, we will evaluate several reliability, availability and performance aspects of our prototype.

Block type	Affected	Recovery time
Direct (actual data)	0.000039 %	28 ms
Single indirect	0.020000 %	157 ms
Double indirect	0.979727 %	6688 ms

Table 2.1: Recovery time after corruption of various data block types in a 100 MB file. For each block type the table lists: (1) the percentage of the file affected when a block of this type is corrupted, and (2) the recovery time measured after corrupting a block of this type.

2.6.1 Test Setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4 GB RAM, and four 500 GB 7200RPM Western Digital Caviar Blue SATA hard disks (WD5000-AAKS), three of which were connected to separate ICIDU SATA PCI EXPRESS cards. We ran all tests on 8 GB test partitions at the beginning of the disks. In experiments where Loris is compared with MFS, both were set up to work with a 32 MB buffer cache.

2.6.2 Evaluating Reliability and Availability

To evaluate the reliability and availability of Loris, we implemented a fault injection block driver that is capable of simulating both fail-partial failures (by corrupting specific data blocks) and fail-stop disk failures (by returning an EIO on all requests).

We first present an evaluation of the ability of Loris to perform on-the-fly data recovery. Rather than just showing that our prototype detects corruption, we illustrate how file-awareness helps in reducing the recovery time. We then present an evaluation of availability under unexpected failures. We show two cases of graceful degradation, both of which cannot be done by block-level RAID implementations.

On-the-Fly Recovery

Our recovery measurements were gathered using a series of fault injection tests. The test file system consists of a single 100 MB file, mirrored over a two-disk Loris installation. The test workload is generated by a user application that issues read requests for specific data ranges in the file. These read requests get forwarded through the stack to the fault injection driver.

The driver corrupts three types of file blocks in three different scenarios: (1) a random direct data block, (2) a random single indirect block, and (3) the double indirect block. In all cases, the driver returns back corrupt data block(s) to the

physical layer. Upon detecting a checksum violation, the physical layer responds to the read request with a checksum error.

The logical layer, on being notified of a checksum error, performs on-the-fly recovery using the redundant copy, and restores lost data onto the corrupt physical layer immediately. Table 2.1 shows the recovery time for various corruption cases. As can be seen, the recovery time is proportional to the amount of data lost within a file.

Graceful Degradation

Our test file system for graceful degradation consists of a collection of 12,000 32 KB files, organized uniformly across 100 directories. We created the test file system on a Loris installation with three disk drives. The number and size of files were chosen to minimize the effect of caching, and the directory layout minimizes the effect of our linear file name lookup.

With this setup, we evaluated graceful degradation with two different file layout schemes, which we will detail shortly. However, in both cases, all directories are mirrored across the three disk drives, and all files are positioned in a fault-isolated manner. Directory replication is done by having the naming layer assign the RAID 1 policy to all directories. Fault-isolated file placement is done by storing each file, in its entirety, in at least a single physical module (as opposed to striping it across all modules). The ease with which we were able to provide these functionalities highlights the flexibility of a file-oriented stack.

Our workload is generated by a program that randomly picks a file and performs a 32 KB read, followed by a 32 KB write, overwriting the entire file. The program considers each open-read-write-close sequence as a single operation, and keeps track of the total number of successful operations.

Figure 2.5 illustrates graceful degradation under no replication. The files in this configuration are uniformly distributed across the three disk drives, that is, each corresponding physical module is responsible for serving a third of file requests. This is made possible by having the logical layer rotate file creation requests between the three physical modules. For instance, the create request for file 1 is forwarded to physical module P1, 2 to P2, 3 to P3, 4 again to P1, and so on.

As it can be seen, when the first fault occurs, the availability drops by roughly 33 %. This is expected since a third of the files are serviced by each physical module, and a device failure renders files serviced by that corresponding module inaccessible. A second failure results in another 33 % drop in availability. At this point, the directory hierarchy remains navigable (because directories are replicated on all drives), and the system continues serve requests that can be satisfied using the last disk drive. It can also be seen that the number of successful requests per second stays unaffected. The sharp drop in performance every thirty seconds

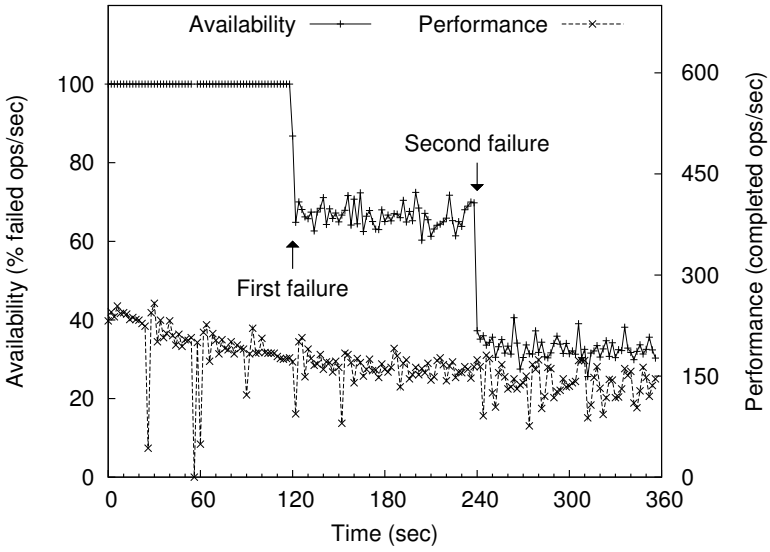


Figure 2.5: Graceful degradation case 1: Unreplicated files. The figure shows how Loris exhibits graceful degradation under unexpected failures. Each disk failure results in a third of files being inaccessible since files are not replicated. But the system continues to survive with an availability of around 33 % even after two disk failures.

is due to synchronous data and metadata flush during a sync.

Figure 2.6 shows graceful degradation, with files protected against a single disk failure. This case further illustrates the advantages of file-level RAID. A block-level RAID implementation would typically use RAID levels 1, 4 or 5 (two data and one parity) to protect against single disk failures. We used RAID 1 and we chose a layout that supported graceful degradation. It should be noted that the same technique can also be used with other RAID levels in a file-level RAID.

With three disks, there are three possible combinations that can be chosen to protect a file against a single disk failure: D1-D2, D1-D3, and D2-D3. Our logical layer rotates files across these combinations. For instance, the first file is mirrored on disks D1-D2, second on D1-D3, third on D2-D3, fourth again on of D1-D2, and so on.

As it can be seen in Figure 2.6, the first failure has no effect on the system, as it would be with block-level RAID, since every file is protected against a single disk failure. A second failure would result in a block-level RAID implementation shutting down. In our case, we can see that the availability drops only by a third. All the files that were mirrored across the two failed disks are now inaccessible. Thus, even with just a single functional disk, Loris is able to maintain an availability close to 66 %.

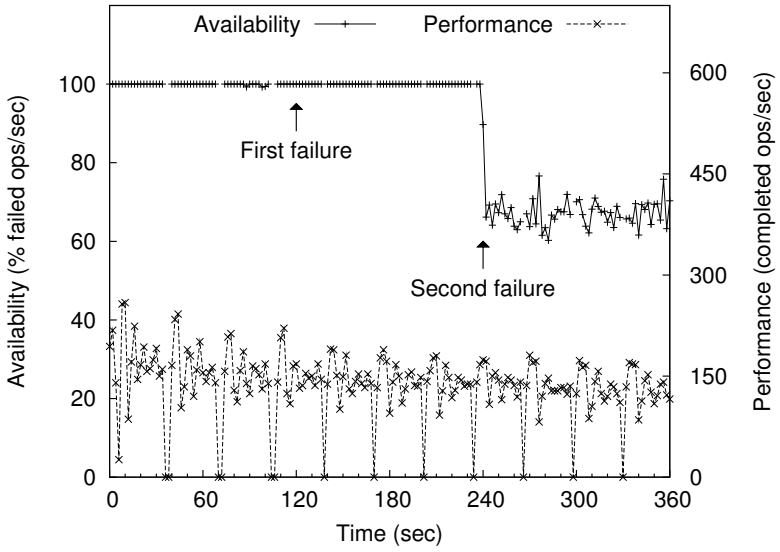


Figure 2.6: Graceful degradation case 2: Rotated mirroring of files. The figure shows how Loris exhibits graceful degradation with files protected against a single disk failure. Each of the three possible pairs holds a third of the files redundantly. As a result, the system continues to serve two-thirds of all its files, with an availability of 66 %, even under two disk failures.

We would like to point out that these techniques are complementary to higher levels of protection, that is, they can be used in combination with RAID 6 techniques, for instance, to build a file-level RAID array with extremely high reliability and availability. Furthermore, these techniques can be customized on a per-file basis, with different files using different levels of protection.

2.6.3 Performance Evaluation

In this section we present the performance evaluation of Loris. We first present an evaluation of the overhead of two important aspects of our new architecture: the parental checksumming scheme as implemented in the physical layer, and the process isolation provided by splitting up our stack into separate processes. We then present an evaluation of our file-level RAID implementation.

Checksumming and Process Isolation

We now evaluate parental checksumming and process isolation using two macrobenchmarks: (1) PostMark, configured to perform 20,000 transactions on 5,000 files, spread over 10 subdirectories, with file sizes ranging from 4 KB to 1 MB,

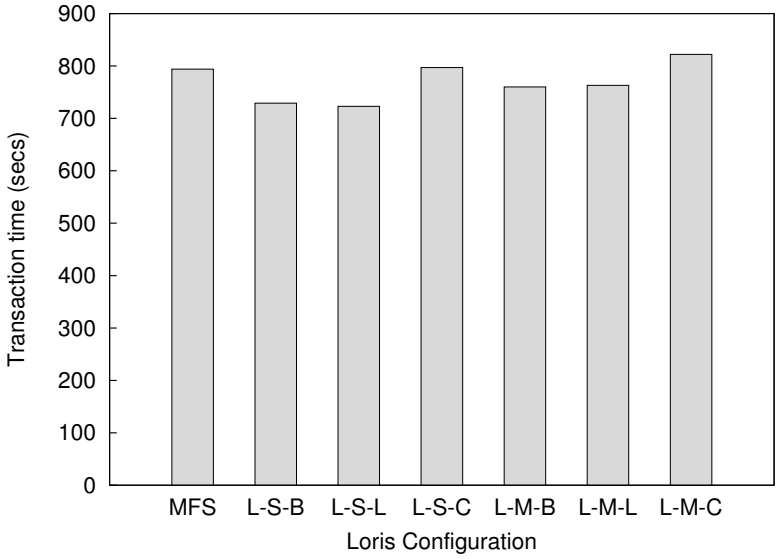


Figure 2.7: Transaction time in seconds for PostMark under MFS and various Loris Loris (L) Single (S) and Multiprocess (M) configurations, without any checksumming features (B), with just parental checksumming layout (L), and with checksum computation (C).

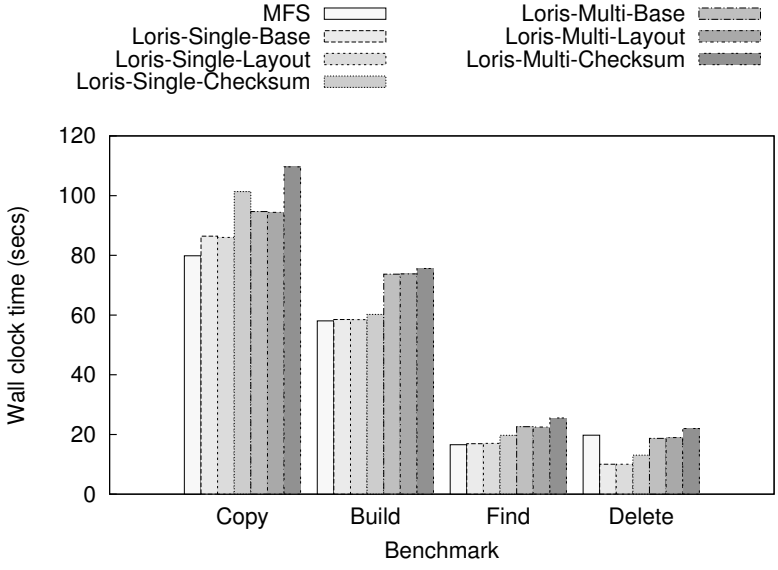


Figure 2.8: Wall clock time in seconds for Applevel benchmarks under MFS and various Loris configurations.

and read/write granularities of 4 KB, and (2) an application-level macrobenchmark, which we will refer to henceforth as *applevel*, consists a set of very common file system operations including copying, compiling, running find and grep, and deleting.

We tested three checksumming schemes: no checksumming, checksumming layout only, and full checksumming. In the layout-only scheme, the CRC32 routine has been substituted by a function that always returns zero. This allowed us to measure only the I/O overhead imposed by parental checksumming. We ran these three schemes in both the single-process and the multiprocess Loris versions, yielding six configurations in total.

Figures 2.7, 2.8 show the performance of all six new configurations, compared to the MFS baseline. Loris outperforms MFS in most configurations with PostMark. This is primarily due to the fact the delete algorithm used by Loris performs better than MFS, as seen in the applevel delete benchmark in Figure 2.8.

Looking at the applevel results, it can be seen that the single-process case suffers from an overhead of about 8 % compared to MFS during the copying phase. This overhead is due to the difference in caching logic between Loris and MFS.

The block-level buffer cache in MFS makes it possible to merge and write out many physically contiguous files during sync periods. Since Loris has a file-level cache, it is unaware of the physical file layout and hence might make less-than-optimal file evictions. In addition, our prototype also limits the number of files that can be written out during a vectored write call, to simplify implementation. These two factors result in a slight overhead, which is particularly noticeable for workloads with a very large number of small files.

Since the copy phase involves copying over 75,000 files, of which a significant percentage is small, there is an 8 % overhead. Even though the overhead is small, we plan on introducing the notion of *file group identifiers*, to enable the passing file relationship hints between layers. The cache layer could then use this information to evict physically contigous files during a vectored write operation. This and several other future optimizations should remove this overhead completely.

Another important observation is the fact that in both single-process and multiprocess configurations, the checksum layout incurs virtually no overhead. This means that the entire parental checksumming infastructure is essentially free. The actual checksum computation, however, is not, as illustrated by a 7 % overhead (over no checksum case) for PostMark, and 3-19 % overhead in applevel tests.

It should be noted that with checksumming enabled, *every* file undergoes checksum verification. We would like to point out that with per-file policy selection in place, we could reduce the overhead easily, by either checksumming only important file data, or by adopting other lightweight verification approaches as opposed to CRC32 (such as XOR-based parity). For example, we could omit checksumming compiler temporaries and other easily regeneratable files.

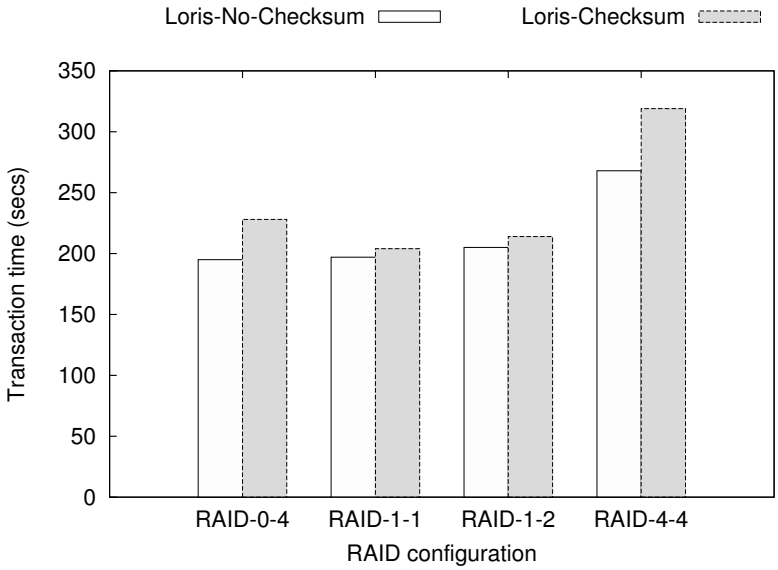


Figure 2.9: Transaction time in seconds, under PostMark, for different RAID levels. Each column RAID X-Y shows the performance of RAID level X in a Y-disk configuration.

We also see that the multiprocess configuration of Loris suffers consistently, with an overhead ranging between 11-45 %. This contradicts the results from PostMark, where the multiprocess case has an overhead of only about 3 % compared to the single-process case. This is again due to the fact that the applevel benchmark has a large number of small files compared to PostMark. Data copying and context switching overheads constitute a considerable portion of the elapsed time when small files dominate the workload. With large files, these overheads are amortized over the data transfer time. We confirmed this with separate microbenchmarks, not shown here, involving copying over a large number of small files.

File-Level RAID

In this section, we evaluate our RAID implementation. We test two RAID 1 configurations: (1) RAID 1 on a single disk, and (2) RAID 1 with mirroring on 2 disks. The RAID 0 and RAID 4 implementations use all four disks, with RAID 0 configured to use 60 KB stripe units, and RAID 4 80 KB stripe units for all files. These stripe sizes align full stripe writes with the maximum number of blocks in a vectored write request (240 KB).

We ran PostMark in a different configuration compared to the earlier benchmark—

20,000 transactions on 60,000 files, distributed across 600 directories, with file sizes ranging from 4 KB to 10 KB. Small-file workloads are challenging for any RAID implementation since they create lots of partial writes. We chose this benchmark to evaluate how our file-level RAID implementation handles small files.

The most important observation from the PostMark results shown in Figure 2.9 is that RAID 4 is much slower than RAID 1 with mirroring. The three main reasons for this slowdown are as follows. (1) RAID 4 suffers from the partial-write problem we mentioned earlier. It is important to note that such partial writes would translate into partial stripe requests in a block-level RAID implementation. (2) Parity computation in RAID 4 is expensive compared to mirroring in RAID 1. Both block and file RAID implementations share these two problems. (3) Our implementation of RAID 4 negates the advantages of vectored writes for small files.

To illustrate this problem, consider a vectored write request at the logical layer. The algorithms used to process this request in our implementation are very similar to a block-level RAID one. Each request is broken down into individual file requests, which are further divided into constituent stripe requests, and each stripe request is processed separately. In our current prototype, we implemented sequential processing of stripe requests. Thus, if the workload consists of many small files, a vectored write gets translated into single writes for each file, negating the benefit of vectoring the write request.

The solution to all the aforementioned problems is simple in our case. Parity amortizes the cost of redundancy only when write requests span multiple stripe units. Thus, we are better off using RAID 1 for small files. As our RAID implementation is file-aware, we can monitor and collect file read/write statistics, and use it to (re)assign appropriate RAID levels to files. Matching file access patterns to storage configurations is future work.

2.7 Conclusion

Despite dramatic changes in the storage landscape, the interfaces between the layers and the division of labor among layers in the traditional stack have remained the same. We evaluated the traditional stack along several different dimensions, and highlighted several major problems that plague the compatibility-driven integration of RAID algorithms. We proposed Loris, a file-level storage stack, and evaluated both reliability and performance aspects of our prototype.

Chapter 3

Flexible, Modular File Volume Virtualization in Loris

Abstract

Traditional file systems made it possible for administrators to create file volumes, on a one-file-volume-per-disk basis. With the advent of RAID algorithms and their integration at the block level, this "one file volume per disk" bond forced administrators to create a single, shared file volume across all users to maximize storage efficiency, thereby complicating administration. To simplify administration, and to introduce new functionalities, file volume virtualization support was added at the block level. This new virtualization engine is commonly referred to as the *volume manager*, and the resulting arrangement, with volume managers operating below file systems, has been referred to as the *traditional storage stack*.

In this paper, we present several problems associated with the compatibility-driven integration of file volume virtualization at the block level. In earlier work, we presented Loris, a reliable, modular storage stack, that solved several problems with the traditional storage stack by design. In this paper, we extend Loris to support file volume virtualization. In doing so, we first present "File pools", our novel storage model to simplify storage administration, and support efficient file volume virtualization. Following this, we will describe how our single unified virtualization infrastructure, with a modular division of labor, is used to support several new functionalities like 1) instantaneous snapshoting of both files and file volumes, 2) efficient snapshot deletion through information sharing, and 3) open-close versioning of files. We then present "Version directories," our unified interface for browsing file history information. Finally, we will evaluate the infrastructure, and provide an in-depth comparison of our approach with other competing approaches.

3.1 Introduction

Over the past few decades in the evolution of file and storage systems, storage virtualization techniques have played a crucial role in improving efficiency and manageability. Traditional file systems provided the first layer of virtualization. File systems were used to create hierarchies of files and directories, also referred to as file volumes[1], on dedicated disk drives. Disks were small enough that administrators could create one file volume per logical unit (per user or per project for instance), and apply administrative policies on these file volumes. As disks grew larger, administrators were forced to use a single file volume across all users to improve storage efficiency. Thus, this "one file volume per device" bond complicated administration, as administrators could no longer use file volumes as the unit of administration.

Volume managers [99] solved this problem by virtualizing file volumes. Similar to RAID algorithms, volume managers were integrated at the block level to retain compatibility with existing installations. The resulting arrangement, with volume managers operating below file systems, has been referred to as the *traditional storage stack*. In this stack, file systems are used to create file volumes on logical disks exposed by the volume manager. Thus, file systems translate file requests to logical block requests. The volume manager transparently maps these logical blocks to blocks on physical devices it manages. As a result, multiple logical disks, and hence multiple file volumes, could now share the same set of physical disk drives, thus improving storage efficiency. Volume managers also simplified administration, as administrators could now create and manage file volumes in logical units.

In this paper, we examine the block-level integration of file volume virtualization, and we highlight several problems along two dimensions: flexibility and heterogeneity. In prior work, we outlined several fatal flaws that plague the traditional stack [8], and presented Loris [9], our complete redesign of the storage stack. Our first prototype, which we refer to henceforth as Loris-V1, had the "one file volume per device" bond, similar to traditional file systems. In this paper, we add support for file volume virtualization to the Loris stack. In doing so, we present *File pools*, a new model for managing storage devices. We show how the new model simplifies management by automating several mundane chores, supports heterogeneous device configurations, and provides file volume virtualization in Loris. We then show how our unified infrastructure, with a modular division of labor among layers, supports 1) instantaneous snapshoting of both files and file volumes, 2) efficient deletion of snapshots by sharing information between layers, and 3) version creation policies, like open-close versioning, on a per-file basis.

[1] Throughout this paper, we will use the term file system to refer to the operating system code that implements a persistent name space of files and directories, and *file volumes* to refer to an instantiation of a file system

We also present *Version directories*, our unified interface for browsing file history information. We will show how version retention policies can be implemented as simple shell scripts.

The rest of the paper is organized as follows. Sec. 2 outlines problems caused by the compatibility-driven, block-level integration of volume managers. In Sec. 3, we present a quick overview of Loris. Sec. 4 presents the design of file volume virtualization in Loris. In Sec. 5, we present the design of efficient file volume snapshoting in Loris. Sec. 6 presents a modular division of labor in Loris that integrates support for both individual file snapshoting, and open-close versioning. Sec. 7 presents our virtual directory interface. We then evaluate Loris using a series of micro and macro benchmarks in Sec. 8. An in-depth comparison of Loris with other systems is presented in Sec. 9. We finally discuss future work in Sec. 10 and conclude in Sec. 11.

3.2 Problems with existing approaches

In this section, we will outline the problems that plague the traditional approach to file volume virtualization. Several commercial and research projects have taken other approaches for virtualizing file volumes. A detailed comparison of our approach with other competing approaches is presented in Sec. 9. We will now present problems along two dimensions, namely, flexibility and heterogeneity.

3.2.1 Lack of Flexibility

An ideal storage stack must 1) provide flexible configuration and management of devices, and 2) support policy assignment at a range of granularities, from individual files or file types, to entire file volumes. In this section, we will highlight how inflexibility in the traditional stack complicates both device management and file management.

Complicated device management

Traditional volume managers used the level of indirection introduced by logical devices to support new functionalities, like file volume snapshoting and cloning. However, this level of indirection also introduced additional administrative operations. Even a simple task, such as adding a new disk to an existing installation, requires a series of steps, at least one for each level in the stack, to be performed by the administrator. This is because, any change in device configuration results in changes, not only in the volume manager's data structures, but also in file system data structures (for instance, any change in the size of a logical disk requires changes to the file system block management data structures), as file systems continue to work with the one file volume per logical disk assumption. Each and

every one of these newly added steps is error prone, and a simple error could result in extensive data loss [17]. An ideal system would allow the administrator to just state the intent, like "add a new disk to an existing installation for increasing storage space," and automate implementation details (like expanding volumes). Traditional block-level volume management fails to meet this requirement.

Coarse-grained file management

Administrators manage data at the granularity of file volumes. For instance, an enterprise administrator could create one file volume per project, and encrypt certain file volumes, while compressing others. Administrators also take snapshots of entire file volumes, and use the snapshot for initiating periodic backups. Thus, at the enterprise level, policy specification at the granularity of file volumes is required. Block-level volume managers can easily provide such policies at a file volume granularity [99].

However, end users tend to associate policies with individual files or file types. The set of files over which a policy must be applied is typically much smaller in number than a file volume. For instance, a user might want open-close versioning on a source file, and no versioning for an object file. Thus, end-users require the ability to specify policies on a per-file basis. Since traditional volume managers operate below a strict block interface, they are semantically unaware, and thus are unable to provide fine-grained file management.

3.2.2 Lack of support for heterogeneous devices

An ideal volume virtualization solution should be designed to work with heterogeneous device types. In this section, we will explain why heterogeneity should be considered a first class citizen during system design. We will show how the traditional approach fails to support devices other than conventional disk drives with a block interface.

Heterogeneity across device families

New devices are emerging with completely new storage interfaces. A common approach to integrating these devices into the traditional stack involves building file systems for each device family [48]. These file systems communicate directly with the device, using device-specific interfaces. However, traditional volume managers can work only with file systems that translate file requests to logical block requests. As a result, the traditional approach of virtualizing file volumes at the block level is not portable across heterogeneous device families.

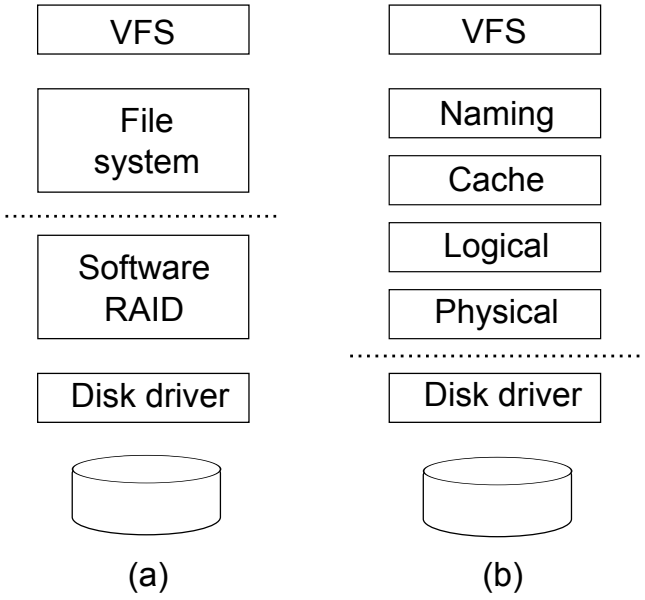


Figure 3.1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

Heterogeneity within device families

Even devices within the same family sometimes differ starkly in their performance characteristics. It is a well known fact that SSDs can be optimized to have different performance characteristics depending on certain firmware design choices [7]. For instance, Intel X25-V SSD provides very good random write IOPS, but its sequential write throughput suffers due to a price/performance tradeoff (only half the channels are populated with NAND). As a result, X25-V provides the same throughput for both large sequential writes, and small random writes [94]. Intel X25-M, on the other hand, provides higher throughput under large sequential writes than small random writes. While it would certainly be beneficial to opt for a log-structured layout on X25-M, it would provide little benefit when a X25-V is used, as it delivers the same throughput for both sequential and random writes. It might even prove to be detrimental due to the unnecessary cleaning overhead.

Thus, device-specific layout requirements create heterogeneity even within device families. Accommodating this kind of heterogeneity is impossible with the traditional stack, where multiple file systems, with even competing layout designs, could share the same physical device. Thus, any layout specific optimizations employed by file systems are rendered futile.

3.3 The Loris storage stack

In prior work, we highlighted several issues that plague the traditional storage stack. We proposed Loris, a fresh redesign of the stack and showed how the right division of labor among layers in Loris solves all problems by design. In this section we provide a brief overview of Loris.

Loris is made up of four layers as shown in Figure 3.1. The interface between these layers is a standardized file interface consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. In addition to file operations, the interface also contains *attribute* manipulation operations—*getattribute* and *setattribute*. Attributes are associated with files and have two purposes in Loris: 1) they enable information sharing between layers, and 2) they store out-of-band file metadata. We will now detail the division of labor between layers in a bottom-up fashion.

3.3.1 Physical layer

The physical layer implements device-specific layout schemes, and provides persistent storage for files and their attributes. It exports a "physical file" abstraction to its client, the logical layer, and by doing so, abstracts away any device-specific interfaces or protocols. Each storage device is managed by a separate instance of the physical layer, and we call each instance, a *physical module*. The physical layer is also in charge of data verification. Being file aware, physical layer implementations can support parental checksumming [55]. By being the lowest layer in the stack, the physical layer verifies both application requests, and requests from other Loris layers alike, thus acting as a single point of data verification.

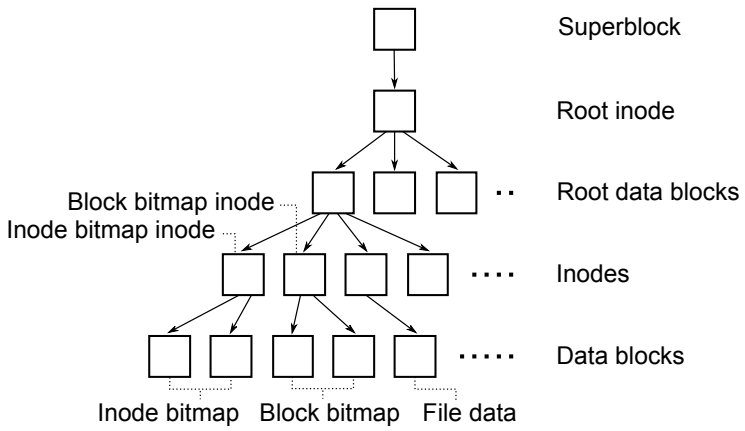


Figure 3.2: Parental checksumming hierarchy used by the physical layer prototype. With respect to parental checksumming, the two special bitmap files are treated as any other files. Indirect blocks have been omitted in this figure.

Our Loris-V1 physical layer implementation was based on the original MINIX 3 file system layout scheme [97], with added support for parental checksumming. Inodes are used to realize the physical file abstraction, and files are referred to using their inode numbers. Each inode stores a fixed number of persistent attributes, as well as 7 direct, one single indirect and one double indirect *safe block pointers*. Each safe block pointer contains a 32-bit block number as well as a checksum of the data block. Single and double indirects also store such pointers. Blocks belonging to both the block bitmap and inode bitmap are checksummed, and their checksums are stored in *block bitmap file* and *inode bitmap file* respectively. Each inode is individually checksummed, and these checksums are stored in the *root file*. The checksum of the root inode itself is stored in the superblock. Thus, the resulting parental checksumming hierarchy, as shown in Figure 3.2, ensures that all blocks, both data and metadata, are verified without any exceptions.

3.3.2 Logical layer

The logical layer's responsibility is to combine multiple physical files and provide a virtualized *logical file* abstraction. It also supports RAID algorithms on a per-file basis. From the point of view of the cache layer, the logical layer's client, a logical file appear to be a single, flat file. Details such as the RAID algorithm used, and the physical files that constitute a logical file, are all abstracted away by the logical file abstraction.

The central data structure in the logical layer is the *mapping file*. The mapping file stores an array of entries, one per logical file, each containing the *configuration information* of that file. This configuration information is 1) the RAID level used, 2) the stripe size used, and 3) the set of physical files that make up the logical file, each specified as a physical module ID and inode number pair. For instance, a file mirrored on two devices could have the following configuration information in its mapping entry: F1=<raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I1, D2:I2>>. The entry tells the logical layer that this file is a RAID1-type file, and inodes I1 on module D1, and I2 on module D2, form the physical files that store F1's data. The mapping file is itself a logical file with a static configuration. The same static inode number is reserved on all physical modules, and the mapping file is mirrored across all these physical files for improved reliability.

3.3.3 Cache and naming layers

The cache layer's responsibility is to provide data caching. Our Loris-V1 cache layer is file aware, and it performs data readahead and eviction on a per-file basis. The cache layer uses a static set of buffer pages to hold cached data.

The naming layer acts as the interface layer. Our Loris-V1 naming layer implements the traditional POSIX interface, and translates virtual file system (VFS) requests into corresponding file operations. All POSIX semantics are confined to the naming layer. For instance, none of the layers below the naming layer know about directories, or the format of directory entries. All other layers treat directories as regular files. The naming layer uses the attribute infrastructure in Loris to store POSIX attributes. The naming layer is also in charge of assigning a unique *file identifier* for each file at file creation time. This identifier is passed as a parameter in all file and attribute operations to identify the target file.

3.4 File volume virtualization in Loris

Like traditional file systems, Loris-V1 does not support virtualized file volumes. However, unlike traditional file systems, the logical layer supports RAID algorithms, and hence can work with multiple devices. Thus, Loris-V1 has a "one file volume per set of devices" bond. In this section, we detail the design and implementation of file volume virtualization in Loris. We first present *file pools*, our new storage model for simplifying and automating the management of devices. We then describe changes to the infrastructure for supporting file volume virtualization.

3.4.1 File pools: Our new storage model

As we mentioned earlier, the standardized file interface above the physical layer abstracts away device-specific details from the logical layer. Thus, from the point of view of the logical layer, each physical module is a source of physical files. Thus, multiple physical modules can be combined together to form a collection of files we call a *file pool*.

Simplified device management

File pools are the unit of storage management. A Loris installation can have one or more file pools. Administrators create a file pool by specifying the set of devices that form the pool. Each device can be a part of only one file pool. Multiple file pools can be created to provide performance isolation for each pool. For instance, an enterprise administrator could create two file pools, each having its own dedicated set of devices, to host the departmental file server and web server.

New devices can be added to, and old devices be removed from, existing file pools. Addition/removal of devices to/from a file pool is completely automated. When a device is added to a pool, a device-specific physical module is started. This new physical module registers itself with the logical layer as a new source of files. Once registration is complete, the logical layer can start creating new

files on this module. Thus, unlike traditional volume managers, adding a new device to an existing file pool is a single step process, and space on the newly added device is immediately available for use.

Any device can be removed from a file pool by just moving all the files on that device to a spare device, or in some cases, even distributing the files among other existing devices. Since the logical layer has complete knowledge of the file-device mapping, supporting this is trivial. Furthermore, since the logical layer is file aware, copying file data ensures that only live data is moved over to the spare disk. This is a huge benefit compared to block-level volume managers, which do a block by block copy of the entire disk due to the absence of block liveliness information [87].

Supporting heterogeneous installations

As mentioned earlier, when a device is added to a file pool, a device-specific physical module is started. Since the physical module exposes a physical file abstraction to its clients, it can completely abstract away heterogeneous device interfaces. As the physical module is in charge of providing device-specific layout, it is possible to support different layout schemes for different devices even within the same device family. Thus, the file pool model permits pairing devices with customized physical modules, thereby exploiting heterogeneity both within and across device families.

Thin provisioning with file pools

We will describe support for file volumes in detail in the next section, but we would like to point out now that file pools also make thin provisioning [26] of file volumes possible. Thin provisioning refers to the ability to create dynamically filled, sparse file volumes. This ability can be used as a planning tool to determine new storage requirements, and hence derive a storage budget. With the file pool model, storage space need not be reserved for file volumes ahead of time. As a result, administrators can create multiple file volumes without committing physical storage space. As users create files and directories in these file volumes, storage space is automatically allocated from the set of physical modules that constitute the file pool. As we will see later, snapshots and clones also utilize this functionality. Multiple snapshots of a file volume can coexist together, but most files and data blocks will be shared among snapshots. Thus, file pools provide natural support for thin provisioning.

3.4.2 Infrastructure support for file pools

We extended the logical layer to support the new storage model. The new logical layer can be considered to be made up of two sublayers, namely, the *volume*

management/RAID sublayer, and the *file pool sublayer*. The file pool sublayer sits below the volume management sublayer and provides device management services, like creating and deleting file pools, adding and removing devices from existing file pools, etc.

The file pool sublayer is also responsible for satisfying file allocation requests from the shared pool of files it manages. It can employ several algorithms for satisfying file allocation requests. For instance, it could maintain a utilization summary of each physical module and provide load leveling, or it could monitor file access patterns and provide workload-aware file allocation. Our file pool implementation just rotates file allocation requests across physical modules. Exploiting device-specific characteristics, and matching device types with file types is a part of ongoing research.

The volume management sublayer operates above the file pool sublayer. We will describe the design details of this sublayer in the next subsection, but it suffices to say now that it supports RAID and volume management algorithms. It utilizes the allocation services of the file pool sublayer to satisfy file allocation requests. For instance, when a new file create request arrives at the logical layer, it is forwarded to the volume management sublayer. The volume management sublayer makes a file create request to the file pool sublayer, passing the number of physical files required (depending on the RAID type chosen for the file) as a parameter. The file pool sublayer allocates the required number of physical files and returns physical module–inode number pairs, which the volume management sublayer then records in its data structures.

3.4.3 Infrastructure support for file volume virtualization

As described earlier, the mapping file in the logical layer stores configuration information for each logical file, and multiplexes file requests across physical files. We decided to extend the logical layer to support multiple virtualized file volumes, since it was a natural extension to the logical layer's data structures.

In the new infrastructure, multiple file volumes can be created in a single file pool, and multiple files can be created in each file volume. As a result, each file is now referenced in Loris using a pair of identifiers, namely, a new *volume identifier*, and the traditional file identifier. The logical layer assigns a new volume identifier to each file volume during volume creation time. Associated with each file volume is a *volume index* file. The format of this file is identical to the original mapping file. Each volume index file contains an array of entries, one per logical file belonging to that volume, and each entry contains contains configuration information, similar to the mapping file's configuration information we described earlier. The volume index file is also mirrored on all physical modules belonging to the file pool for improving reliability.

Since volume index files are created during volume creation, the configuration

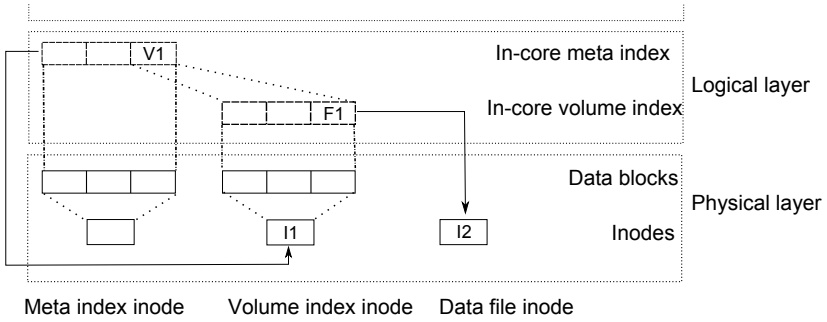


Figure 3.3: The figure shows the relationship between meta index and volume index, the two datastructures that support file volume virtualization in Loris. The meta index file itself is a static file, and static inode is reserved to store its data(an array of file volume metadata entries). The file volume metadata entry for volume V1 shown in the figure could be <V1, REGULARVOL, volume index configuration=<raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I1>>. Thus, inode I1 in physical module D1 is used to store the volume index file data (an array of logical file configuration entries) for file volume V1. The logical file configuration entry for file <V1, F1> could be <raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I2>>. Thus, inode I2 in physical module D1 is used to store file data.

information of the volume index file itself is not static, that is, the volume index file can use inodes with different inode numbers on different physical modules. Hence, this configuration information is stored in the *meta index* file, with other file volume metadata. Each file pool has only one meta index file, which is also mirrored on all physical modules for improving reliability. This file contains an array of entries, one per file volume, containing *file volume metadata*. This metadata consists of 1) configuration information for the file volume's volume index file, 2) the type of the file volume, and 3) the volume id for this file volume. Thus, while the volume index file tracks files within a volume, the meta index tracks file volumes themselves. When the new logical layer receives a call to perform any file operation, it uses the volume identifier to first retrieve the volume metadata from the meta index file. After this, it uses the file identifier to retrieve the target file's configuration information, following which, it performs the requested operation. Thus, these two data structures make it possible for multiple file volumes to share a file pool, as shown in Figure 3.3, effectively breaking the one file volume per set of devices bond.

3.5 New functionality: file volume snapshoting in Loris

Loris supports an extremely flexible snapshoting facility. snapshoting is efficient and instantaneous in Loris. We added a new *snapshot* operation to the standard-

ized file interface described earlier. The operation carries a parameter, which is either the target file identifier for an individual file snapshot, or the file volume identifier for a file volume snapshot.

3.5.1 Division of labor

Space-efficient snapshoting requires fine-grained, block-level data sharing to avoid making unnecessary copies of unchanged blocks. After investigating several possibilities, we assigned the responsibility for providing data sharing to the physical layer. This labor assignment maximizes storage efficiency without sacrificing modularity, as it is possible to support different physical layer implementations, with different mechanisms for data sharing, without affecting the logical layer algorithms. Thus, each physical layer must provide support for physical file snapshoting. In this section, we will describe two such physical layer implementations—a copy-based physical layer that lacks storage efficiency, but is extremely simple to implement, and a copy-on-write physical layer that supports fine-grained data sharing.

With individual file snapshoting and data sharing mechanism provided by the physical layer, the logical layer acts as a policy engine. It decides when a snapshot operation should be invoked on which physical file, and supports file volume snapshoting using individual file snapshoting provided by the physical layer. In this section, we will describe the logical layer data structures that support file volume snapshoting after describing the two physical layer implementations.

3.5.2 Physical layer(1): Copy-based snapshoting

To implement copy-based snapshoting, we retained the layout design of the Loris-V1 physical layer, and we added support for the new snapshot operation. In both copy-based and copy-on-write-based physical layers, we distinguish between two types of inodes, namely, *current inodes*, and *snapshot inodes*. Current inodes can be considered to be the active version of a file which is used to satisfy normal read/write requests. Snapshot inodes, on the other hand, are read-only, historical versions, that act as point-in-time snapshots of a current inode. A snapshot inode is created as a result of a snapshot operation on a current inode. We will now describe how snapshot creation and deletion work in the copy-based physical layer.

Snapshot creation

Since the physical layer is responsible for storing both data and attributes (POSIX attributes for instance), it must preserve their old values after a snapshot. So, the copy-based physical layer performs the following steps during a snapshot call. It

first retrieves the inode corresponding to the inode number passed in as a parameter to the snapshot call, which we will refer to henceforth as the target inode. It then allocates a new inode, and copies over all the attributes from the target inode to the new inode. Following this, data belonging to the target inode is also copied over, allocating new data blocks during the process, to the new inode. Thus, after a snapshot operation, the new inode and target inode are independent copies, *not sharing* any data blocks. Finally, the new inode number is returned back to the the logical layer. From here on, the target inode becomes a snapshot inode, and the new inode becomes the current inode. It is important to note here that the level of indirection provided by the logical layer makes it possible to switch inodes without changing the file identifier. As a result, higher layers in the stack, like the naming layer, can continue using the same file identifier even after a snapshot operation.

Snapshot deletion

Deletion of a snapshot inode is a trivial operation. Since no data is shared between snapshots, the deletion operation deallocates all data, single and double indirect blocks, and then the inode itself, by marking them free in their corresponding bitmaps. Thus, while copy-based snapshoting suffers from inefficient storage utilization, its conceptual simplicity makes it a good mechanism for some personal and enterprise computing environments, where accesses to small files dominate the workload.

3.5.3 Physical layer(2): Copy-on-write-based snapshoting

Our copy-on-write layout is a natural extension of the Loris-V1 layout. There are two major requirements for supporting copy-on-write-based snapshots as we will see in this section. The first requirement is that, for each data block, we need to identify if the block is shared with a previous snapshot. This is required to be able to perform a copy-on-write operation *only* when required. Second, for each snapshot inode, we need to know the chronological successor and predecessor to provide efficient snapshot deletion.

To support the former, we changed the definition of a safe block pointer. As we mentioned earlier, both inodes and indirect blocks contain a number of safe block pointers, and each safe block pointer contains a 32-bit block number–checksum pair. On an installation with a 4-KB block size, the largest disk size that can be supported with a 32-bit block number is 16-TB. For our prototype, we borrowed a bit from the block number, and the resulting safe block pointer contains a 31 bit block number, a 1- bit status field, and the block checksum. The resulting layout can now support a maximum disk size of 8-TB. Setting the status bit marks a data block as being newly allocated in this snapshot, and hence unshared with

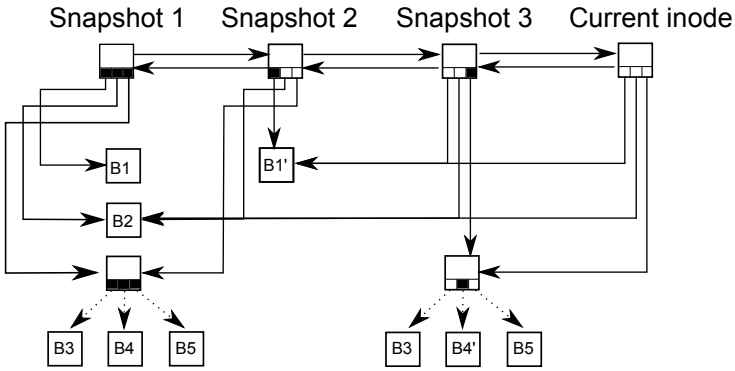


Figure 3.4: The figure shows physical layer data structures after performing the following operations: 1) Create file (write B1, B2, B3, B4 and B5) 2) Snapshot file 3) Modify B1 to B1' 4) Snapshot file. 5) Modify B4 to B4'. 6) Snapshot file. A black rectangle represents a status bit that is set (an unshared block), and a white rectangle represents a cleared status bit(a shared block).

the previous snapshot. Clearing the status bit marks a data block as shared with the previous snapshot. We adopted this approach of borrowing a bit only for reducing the implementation effort. It is always possible to make the safe block pointer larger and overcome this space limitation. To maintain a chronological relationship between snapshots, we added two new fields to the inode, the *previous snapshot* and *next snapshot*. These fields are the inode numbers of the previous and next snapshot inodes respectively, and they link snapshots together in a bidirectional list.

Snapshot creation

When the copy-on-write physical layer receives a snapshot request, it performs the following steps. It first allocates a new inode, and copies over all the attributes from the target inode, just like the copy-based snapshoting approach. However, unlike the copy-based approach, it then copies over only the safe block pointers from the target inode, instead of allocating new blocks. While copying over the safe block pointers, it clears the status bit in each pointer to indicate that the data blocks are shared between the new and snapshot inodes. The physical layer then adds both inodes to the bidirectional list of snapshots by setting the previous and next snapshot fields. Finally, it returns back the new inode number to the logical layer. From here on, the target inode becomes a snapshot inode, and the new inode becomes the current inode. Figure 3.4 illustrates the snapshot operation with an example.

Copy-on-write mechanism

When the physical layer receives a write request, it first retrieves the target inode. For each block being written, the physical layer then retrieves the corresponding safe block pointers from either the inode, or from one of the indirects. If the status bit in the safe block pointer is cleared, the data block is shared with the previous snapshot. Hence, the physical layer allocates a new block, and the data is written out to this new location. However, if the status bit is set, no allocation happens, and the data is written to the block address contained in the block pointer.

If a new data block was allocated, the physical layer must update its corresponding block pointer in either the inode, or one of the indirects, to reflect 1) the new location of this data block, and 2) the new unshared state of the block by setting the status bit. If the safe block pointer is in the inode, updating this information is trivial, as shown for snapshot 2 in Figure 3.4. However, if the block pointer is in an indirect block, then the physical layer must allocate a new indirect, since the old one is being referenced by the previous snapshot.

In Figure 3.4, snapshot 3 illustrates the indirect block update process with an example. First, the old indirect block pointed to by the inode is read in. Following this, a new indirect block is allocated, and it is populated with safe block pointers from the old indirect block. During this process, the status bits in the safe block pointers are cleared. Following this, the safe block pointer for the newly allocated data block is updated in the indirect block. Finally, the safe block pointer for the indirect itself is updated in the inode. As shown in the figure, by clearing the status bit for all shared data pointers, we postpone block allocation until the very last moment, thereby providing efficient data sharing. As allocation of blocks takes place in a dynamic fashion, the amount of space taken by a snapshot is proportional to the amount of data overwritten.

Snapshot deletion

Deleting a snapshot inode is more complicated, since blocks pointed to by an inode could be shared with other snapshot inodes. A block can be freed only if it is not shared with any other inode. We use two facts to help us make this decision. For any given snapshot inode,

1. any data block not shared with the immediate predecessor is also not shared with any other predecessor.

2. any data block not shared with the immediate successor is also not shared with any other successor.

Thus, a block that is not shared with the immediate predecessor and successor snapshots are blocks that are unshared with any other snapshots, and hence by definition, are blocks that can be deleted. We can easily find blocks of the former kind

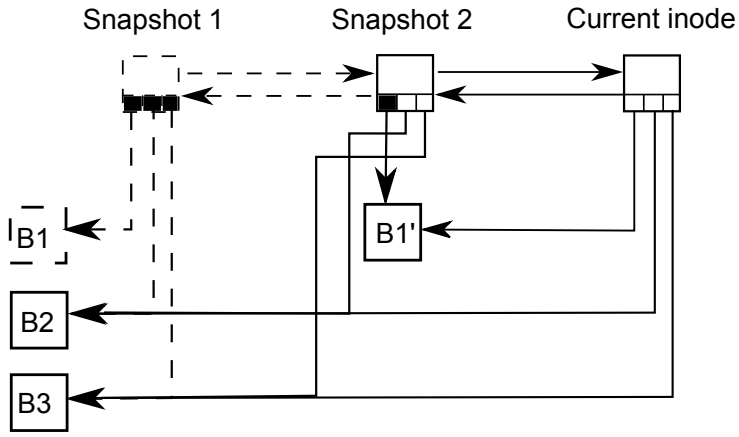


Figure 3.5: The figure shows the physical layer data structures after the following operations are performed: 1) Create file 2) Snapshot 3) Modify B1 4) Snapshot 5) Delete the first snapshot

using the status bits in the safe block pointers associated with the target inode. We can find blocks of the latter kind by reading in the safe block pointers associated with the target's successor inode, and examine their status bits. However, there are two interesting boundary conditions that deserve a special mention.

The first condition is when a file is truncated in the successor. In such a case, some of the successor's safe block pointers would be invalid, as the truncation code zeroes out these block pointers (by setting them to NOBLOCK). The second condition occurs when the target of deletion is the head of the snapshot list. Consider the situation depicted in Figure 3.5. When the first snapshot is deleted, block B1 is freed, but B2 is not freed as it is shared with the second snapshot. After deleting the first snapshot, the second snapshot is at the head of the snapshot list. However, the status bit for block B2 is cleared in the snapshot inode, as B2 was not overwritten during the snapshot lifetime. Thus, if the second snapshot is deleted, B2 will not be freed. Thus, considering both boundary conditions, we adopt the following algorithm for deleting snapshot inodes.

```
for each block offset in inode_being_deleted do
  if pointer_in_successor.block_number = NOBLOCK or
  pointer_in_successor.status = 1 then
    if pointer_in_deleted_inode.status = 1 or
    inode_being_deleted.predecessor = NONE then
      delete the block
    end if
  end if
end for
```

3.5.4 File volume snapshoting in the logical layer

Having explained the mechanism for block sharing in the physical layer, we will now explain the snapshot operation at the logical layer. We will also show how information sharing between the logical and physical layers makes it possible to support efficient snapshot deletion.

In order to support snapshoting, each logical file is associated with a *file epoch number*. This epoch number is stored together with other logical file configuration information in the corresponding volume index file. Each file volume is also associated with a *volume epoch number*. This epoch number is stored together with other file volume metadata in the meta index file. Each file volume metadata entry also contains previous and next snapshot fields. These fields store the volume identifiers of preceding and succeeding snapshot volumes, thus linking snapshots in a bidirectional list, similar to the inode snapshots in the physical layer. As we will see later, version directories utilize this bidirectional linking to enumerate the list of snapshots.

Snapshot creation

When the logical layer receives a request to snapshot a file volume, which we will henceforth refer to as the target volume, it first retrieves the volume metadata from the meta index file. The logical layer then creates a new *snapshot volume*. A snapshot volume is a read-only file volume. No file operations, except read and getattribute, are permitted on any files in a snapshot volume. The type field in the volume metadata indicates whether a volume is a regular or a snapshot volume. The process of creating a snapshot volume involves 1) assigning a new volume identifier, 2) allocating an entry for storing the new volume's metadata in the meta index file. Following this, the logical layer copies over all metadata fields from the target volume entry to the snapshot volume entry. Once this step is completed, both the snapshot and target volumes share the same volume index file. Figure 3.6(b) illustrates the first step with an example.

Next, the logical layer snapshots the volume index file. It does this by making a snapshot call to each physical module, passing in the relevant inode number as a parameter. Each physical module snapshots the inode as explained in the previous section, and returns back a new inode number. The logical layer updates only the target volume's volume index configuration information with this new inode number. At this point, the snapshot volume's configuration points to the old volume index inodes, and the target volume's configuration points to the new inodes, as shown in Figure 3.6(c). Finally, the epoch number in the target volume is incremented by 1 to reflect a completed file volume snapshot operation. Thus, snapshoting is an instantaneous operation, and it involves only a snapshot call to freeze the volume index file.

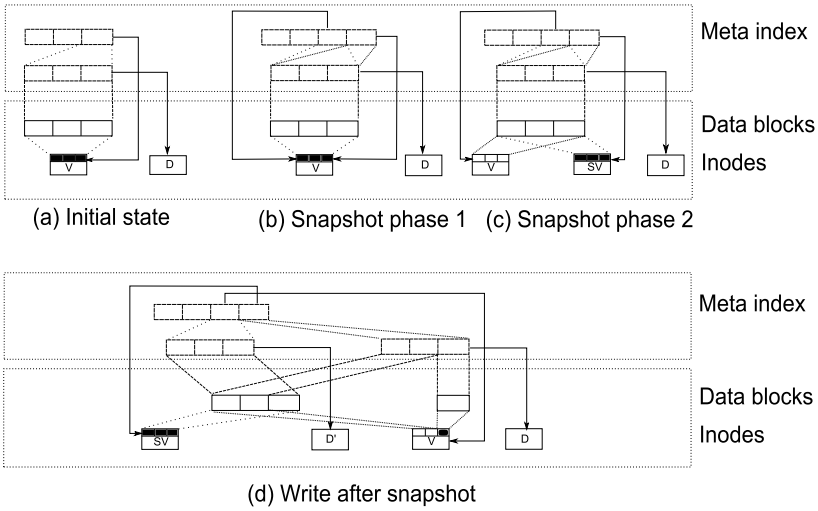


Figure 3.6: The figure shows the different phases of a snapshot operation, and the interaction between logical and physical layer data structures during, and after a snapshot operation. A block labeled D in the figure represents a data file inode, V represents a current/regular volume's volume index inode, and SV represents a snapshot volume's volume index inode. Arrows in the figure connect a file volume metadata entry with the volume index inode, and a logical file configuration entry with its data file inode. Blocks in the logical layer show the logical layer's view of the meta and volume index files, while blocks in the physical layer represent the actual blocks storing data corresponding to those file. Dotted lines between the two layers show the mapping between the two views. The figure is divided into four parts. Part (a) shows the state of the stack before a snapshot. Part (b) shows the first phase of the snapshot operation, where a new snapshot volume is created. Part (c) shows the second phase of the operation, where a snapshot of the volume index file itself has been performed. Part (d) shows a new data file inode being allocated due to a write operation after a snapshot, and the new volume index file pointing to the new data file.

Snapshoting of individual files happens dynamically at the next operation that modifies the data or metadata of the file. When the logical layer receives a write, delete, setattr or truncate operation on a file, it compares the file's epoch number with the corresponding file volume's epoch number. If the file has a smaller epoch number, the logical layer snapshots the file before performing the operation. Read, create and getattr Loris operations do not incur this check, as they do not modify a file or its metadata in any way. For instance, a read request from the application gets transformed into a Loris read operation to retrieve the data, and a Loris setattr operation to update the access time. While the Loris read operation bypasses the check, the setattr operation results in a snapshot being created. Finally, the file's epoch number is set to the file volume's epoch number. Future operations on the

file proceed without snapshoting the file again as the epoch number test fails. The logical layer snapshots a file by calling the snapshot operation on all associated physical modules. The file's configuration information is updated with the new inode numbers returned by the physical modules. When the data block containing this new file entry is written out to the physical layer, the data sharing mechanism in the physical layer ensures that only the current volume index file stores this new configuration. Thus, as illustrated in Figure 3.6(d), any snapshot of a file is reachable through that snapshot volume's volume index.

Efficient deletion support through information sharing

The algorithm for deleting snapshot volumes in the logical layer is very similar to the algorithm for deleting blocks in the physical layer. In the pseudocode for deleting snapshots given below, target snapshot refers to the snapshot volume being deleted.

```
for each file in the target snapshot do
  if file does not exist in the next snapshot or file has been modified in the next
  snapshot then
    if file has been modified in the target snapshot or target snapshot has no
    preceding snapshot then
      call delete on this file's physical modules
    end if
  end if
end for
```

Similar to block deletion, files that are not shared between snapshots are the files that are deletable. A file is not shared by two snapshots if it has been modified between snapshots. As mentioned earlier, a file is modified if the file receives a setattribute, delete, truncate or write operation after a file volume snapshot. Any such modification operation results in a new file entry, with a new configuration information, and an updated epoch number. Thus, unshared files are ones for which file epoch number is the same as the snapshot volume's epoch number. Each such file identified by the algorithm is deleted by making a delete call to each corresponding physical module, which deletes the snapshot inode as mentioned earlier.

It is a well-known fact that file access distribution is heavily skewed, with a very small percentage of files getting a large percentage of accesses [79]. Thus, it is very likely that only a relatively small number of files, and hence file entries, are modified between any two snapshots. The deletion efficiency could be improved significantly if we process only these changed file entries. Thinking about this, we realized that the copy-on-write-based physical layer already stores this information in the form of status bits associated with each block. Thus, we added a new operation which the physical layer could use to communicate a snapshot

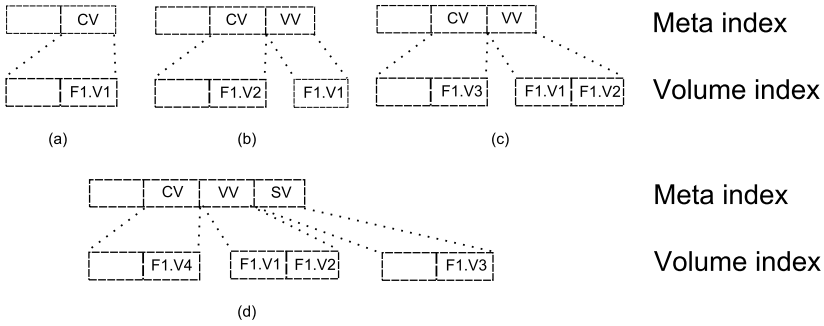


Figure 3.7: The figure illustrates individual file snapshoting using version volumes. Part(a) in the figure shows a current volume(CV) containing configuration information for a file F1. Part (b) in the figure shows the data structures after an individual snapshot of file F1. It shows that a new version volume(VV) has been created, and F1's V1 configuration information is stored as an entry in VV. Part (c) shows the state after another individual file snapshot. Now, VV has two entries, one configuration per snapshot. Part (d) shows the state after the following operations: 1) snapshot volume CV, 2) write F1. A snapshot volume(SV) has been created, and it contains the configuration information for version V3 of F1. It is important to note that VV stores only individual file snapshot history(V1 and V2 of the file). File volume snapshot history (V3 in this case) is recorded by snapshot volumes, SV in this case.

inode's modified block offsets to the logical layer. When the logical layer receives a delete request, it retrieves the file volume configuration information as usual. It then makes a call to retrieve the set of modified file offsets for the snapshot's volume index file. Equipped with this information, the logical layer executes the algorithm mentioned earlier, but only for file entries in these modified offsets thus avoiding a linear scan.

3.6 New functionality: Unifying file snapshoting and version creation policies

In this section, we will describe the infrastructure support for providing per-file snapshoting and open-close versioning. Per-file snapshoting and open-close versioning introduce an interesting problem in the design of file volumes. With both these features, multiple snapshots of a file can be taken between any two file volume snapshots. Each such snapshot requires the configuration information at the time of snapshot to be recorded. In our file volume design, each configuration information is always tracked by a volume index belonging to either a snapshot volume or a current volume. For instance, after the first operation that modifies file data or metadata following a snapshot, the snapshot volume's volume index entry

tracks the old inodes that existed at the time of snapshot, as we already described earlier. However, with individual file snapshots, no volume index is available to track multiple snapshot configurations. Hence, without added support, we lose the ability to access all individual snapshots created between two file volume snapshots.

Solving this problem requires a way to track version history, in the form of configuration information for each file snapshot. Thinking about this, we realized that we could create a new file volume for each file, and use its volume index file to store this configuration information.

3.6.1 Version volumes

To support per-file snapshoting, we define a new volume type, called *Version volume*. The fundamental idea behind version volume is to group all individual file snapshots together in a file volume. Each logical file in Loris can be conceptually seen as being associated with a version volume, and each version volume stores the version history of its parent file. A version volume is linked to its parent file by storing its volume identifier with the file's configuration information. We will now explain how version volumes are created, and how they support per-file snapshoting.

Initially, all files start out without a version volume. A version volume is created during the first individual file snapshot operation. Creating a version volume is similar to creating a regular volume–a new volume identifier is assigned, a new metadata entry is created in the meta-index file, a new volume index file is created, and its configuration information is stored with other details in metadata entry. An important piece of metadata specific to version volumes is the *next version number* field. This field is incremented every time a new file version is added to the version volume. The logical layer also stores the version volume's identifier with the file's configuration information, thereby linking the file with its version volume.

Every individual file snapshot operation proceeds as follows. The target file's configuration information is retrieved from the corresponding volume index file. The version volume's next version number is incremented, and this value is used to determine the version volume's volume index entry where this old configuration information is stored. Following this, a snapshot call is made to all associated physical modules, and the target file's configuration information is updated in the current volume. Figure 3.7 illustrates this with an example.

Figure 3.7 illustrates the interaction between a file volume snapshot and an individual file snapshot. Version volumes are used only to store configuration information for snapshots created by user-initiated per-file snapshots, or auto-generated open-close versioning based snapshots. File snapshots created as a side effect of a file volume snapshot are tracked by the respective snapshot volume's volume index.

While it might appear at first thought that creating a file volume for each file might be expensive, such is not the case due to several reasons. First of all, files start out initially without a version volume. As we mentioned earlier, version volumes are created on-the-fly during the first individual file snapshot operation. Thus, all files that are not modified after a snapshot do not have version volumes. Second, each volume requires space proportional to the number of configuration entries it stores. As version volumes are created on a per-file basis, only files that are snapshotted very frequently end up with version volumes containing many configuration entries. Third, as all individual file versions share unmodified data blocks using the physical layer's copy-on-write functionality, the only information that is not shared between versions is each version's configuration entry, which by itself is very small (roughly 100 bytes) compared to the modified data blocks. Thus, version volumes provide a light-weight mechanism for tracking file history.

3.6.2 Open-close versioning in the naming layer

We will now illustrate how the same infrastructure that supports individual file snapshoting supports open-close versioning as well. The naming layer, being aware of open-close sessions, acts as the policy enforcement layer. It creates new versions of files that have been modified in an open-close session, by making a snapshot call following the close operation. The logical layer processes this snapshot call, as explained earlier, using its version volume infrastructure, and forwards the snapshot call to each associated physical module for snapshoting the inodes. The next setattr, write, truncate or delete operation on this file will result in the physical layer performing block-granular copy-on-write. Thus, as the naming layer only specifies policies, plugging in a different version creation policy, like provenance-based version creation [67], is very simple. The only code change required would be to figure out the exact time and place where a snapshot call must be made. Thus, using a unified infrastructure, we are able to support 1) per-file version creation policies, 2) per-file snapshoting, and 3) file volume snapshoting.

3.7 New functionality: Version directories– a unified interface for browsing history

Snapshoting and versioning systems have provided several different interfaces for browsing file history information. The design requirements we had for our interface were threefold: 1) the interface should be simple and natural to use, 2) applications should be able to access file versions without any modifications, 3) the same interface should be used for accessing individual file snapshots, open-close version based snapshots, and file volume snapshots. We now present *version directories*, our new unified interface for browsing file history information.

3.7.1 Version directories – interface specification

With most versioning systems, users can access file versions by suffixing a file name with a *version specifier*. The version specifier consists of a *syntax token*, which is a special character (like "!" in CedarFS [33]), and a *version sequence number*, which identifies the target version from a list of available file versions. Most snapshoting systems, on the other hand, require users to mount a file volume snapshot, or auto mount the snapshot at a designated mount point. We rejected the mount-based history access as it violated our third requirement.

In our interface, the "@" character acts as the syntax token. Thus, for a file foo, the name foo@N can be used to access the Nth version of foo. Version specifiers can also be used with directories to achieve *version inheritance*. Version inheritance refers to the mechanism by which files and directories are automatically scoped using their parent's sequence number. Version inheritance can be used to simulate a mount-based interface. For instance, if /home/user1 is a file volume, an administrator could create a symbolic link to /home/user1@1 at any location, and scope the entire subtree to the first snapshot.

Yet another interesting feature of this inheritance mechanism is its use in recovering deleted files. Since we do not support name versioning yet, once a file is deleted, its name is removed from its parent directory by the naming layer, the file entry is purged from the current file volume by the logical layer, and the inodes storing file data are deleted by the corresponding physical modules. However, a user could scope the parent directory to a snapshot that has the file name intact, and by inheritance, access the file version at that snapshot. For instance if a file foo has been removed from directory bar, the name bar/foo@2 cannot be used to access the snapshot version of foo, as we do not support name versioning yet. However, the name bar@2/foo could be used to achieve the same effect. As an aside, the name bar@2/foo@1 resolves to the same version as the name bar@1/foo, and the name bar@1/foo@2 resolves the same version as the name bar@2/foo. Thus, multiple scope specifiers can be used in a single path name.

Having described the interface for accessing old versions, we will now describe our interface to enumerate the list of all file versions. Most systems add an explicit library call, that in turn forwards an ioctl to a versioning file system to retrieve such details. Previous research has already suggested that overloading file system semantics improves uniformity when compared to creating new interfaces [32]. In our interface, by suffixing any file or directory name with *just* the syntax token, the user can treat it as a *version directory*. A version directory is a virtual directory, in the sense that its directory entries are created on the fly. Virtual directories meet all the three requirements we mentioned earlier: 1) It is a simple and natural technique, as it overloads a well known file system construct—directories, 2) one can use shell utilities and applications unmodified to access old versions, and 3) every snapshot—irrespective of how it is created—is presented

as a virtual directory entry, thus providing an integrated access interface.

A directory entry enumeration operation on a version directory results in all versions of the target file being displayed to the user as individual file entries in the directory. For instance, a user could perform a "cd foo@", followed by an "ls –l" to view both dynamically generated file names and POSIX attributes of each individual version. Each file name in a version directory has two parts: a type specifier, and a version sequence number. The type specifier identifies whether the version was created by an individual file snapshot (SNAP), an open-close versioning snapshot (VERSION), or a file volume snapshot (VOLSNAP). As an aside, users can also access versions by using directory entries instead of suffixing file names with sequence numbers. For instance, /usr/foo.txt@1, and /usr/foo.txt@/VolSnap_1 resolve to the same file version.

Another advantage of using version directories is the fact that retention policies can be implemented as simple shell scripts. For instance, a script that implements a number based retention policy could access each file as a version directory, and delete oldest N versions using standard UNIX utilities. A landmark based retention policy could diff two versions (diff foo@1 foo@2), and pick versions with minimal changes to discard. Furthermore, different policies can be applied to different files or file types, thus providing highly flexible version management. It is important to note here that only snapshots created by open-close versioning or individual file snapshoting can be explicitly deleted. File snapshots created as a side effect of volume snapshoting can be deleted only by deleting the file volume. Retention scripts can use the type specifier part of the file name to identify deletable versions.

3.7.2 Version directories – implementation details

We will now describe the infrastructure support for implementing version directories. When an applications performs a directory listing operation on a virtual directory, the naming layer needs to enumerate the list of snapshots for the target file. Since the task of tracking snapshots is provided by the logical layer, a new *version stat* call was added to the standardized file interface to communicate this information to the naming layer. The naming layer makes the version stat call, passing in the target file identifier as a parameter.

When the logical layer receives a version stat call, it first pulls up the volume metadata from the meta index file. It then walks through the set of snapshots associated with this volume, using the previous and next snapshot fields we described earlier, and checks each volume index for the target file. For each valid configuration entry, the logical layer populates a new *vstat structure*. Each vstat structure contains several fields, like the volume identifier, volume type, and file identifier. After checking all snapshot volumes, the logical layer retrieves the file's configuration information from its current volume. If the configuration in-

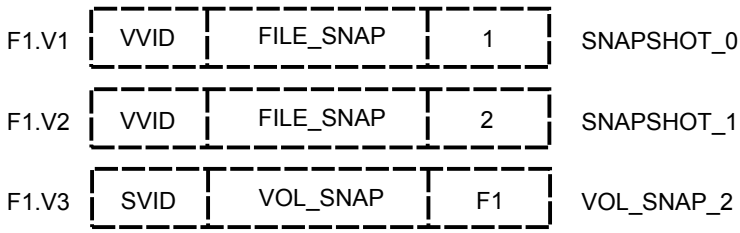


Figure 3.8: The figure shows the vstat structures for file F1 shown in Figure 3.7(d). The columns from left to right contain the volume identifier, volume type, and file identifier for the file version shown to the left of the vstat structure. The names on the right of each vstat structure are the names assigned by the naming layer, for each virtual directory entry, when the file is treated as a version directory.

formation records the presence of a version volume, the logical layer retrieves the version volume's volume index file. For each configuration entry in this volume index, it populates a new vstat structure. After processing all entries, the logical layer returns back the list of vstat structures to the naming layer, as shown in Figure 3.8.

The naming layer uses these vstat structures to build virtual directory entries in the version directory. It uses the volume type field to choose a type specifier, and a zero based counter to assign the version sequence number for each directory entry. When the user access a particular file version using one of these entries, the naming layer uses the version sequence number to retrieve the appropriate vstat structure. It then uses the volume and file identifiers specified in the vstat structure to identify the target file in any file operation.

This approach does have the disadvantage that a version's name might change as other versions are deleted. However, we really do not consider this to be an issue due to two reasons. First of all, as users can easily view POSIX metadata associated with each version, they can identify the target version using its metadata, thus obviating the need for consistent system-generated names. Second, if name-based identification is required, the proper approach would be to enable tagging of individual versions. Once users tag versions with user-friendly names, they can easily identify target versions using their tags. We are working on extending our system to support tagging of both individual file versions and file volume snapshots.

It is important to note in Figure 3.8, that the file identifiers in the first two vstat structures are not F1, the target file's identifier. Since these two versions were created by individual file snapshoting, their configuration information resides in the version volume's volume index. Since the version volume can be accessed as a regular file volume, one could directly access a version by using its position within the volume index as the file identifier. For instance, when the logical layer

gets the identifier pair <VVID, 1>, it first retrieves the volume with identifier VVID, which in our case is the file's version volume. It then uses 1 as the file identifier, and retrieves the first file entry from the volume index, which would be the configuration information for the first file version. Thus, by grouping all file snapshots in a version volume, we are able to use the same mechanism for accessing file snapshots, irrespective of how they are created.

3.8 Evaluation

In this section we will present our evaluation of the Loris prototype which supports all new functionality presented in this paper. We implemented our Loris prototype on the MINIX 3 multiserver operating system [44]. We will first evaluate the overhead of open-close versioning and snapshoting using micro-benchmarks. We will then present an evaluation of the infrastructure using two macro-benchmarks, and show that our file volume virtualization approach has no overhead.

3.8.1 Test Setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4 GB RAM, and one 500 GB 7200RPM Western Digital Caviar Blue SATA HDD (WD5000AAKS). We ran all tests on 8 GB test partitions at the beginning of the disk. Loris was set up to work with a 32 MB buffer cache.

3.8.2 Copy-based and copy-on-write snapshoting comparison

We will first compare the performance of copy-based and copy-on-write-based snapshoting using a custom micro-benchmark. The micro-benchmark stresses file system snapshoting by first creating either 500 1-MB files, or one 500-MB file, in a single file volume. Following this, we perform ten rewrite runs, where we truncate and rewrite all the files, snapshoting the entire file volume after each run. We measure the total time taken to overwrite all the files in each run, and the median of these ten values is shown in Table 3.1. As the file volume is snapshoted, each rewrite run results in a new snapshot of all files being created.

Benchmark	No Snapshot	Copy-based	Copy-on-write-based
500 1-MB files	7.30	17.95	7.31
1 500-MB file	7.50	21.01	7.95

Table 3.1: Time in seconds for file volume snapshoting using copy-based and copy-on-write-based physical layer implementations.

As shown in Table 3.1, file volume snapshoting has very little overhead with the copy-on-write-based physical layer. The copy-based physical layer however has a significant overhead. This is due to the fact that each file is copied over in its entirety after every snapshot operation. As each copy operation reads and writes 1-MB per file snapshot in the first case, and 500 MB in the second case, it causes excessive delay in the mainline write path leading to poor performance.

3.8.3 Open-close versioning evaluation

We will now present an evaluation of our open-close versioning implementation using the same micro-benchmark that was used to evaluate snapshoting, with a minor modification. We no longer snapshot the file volume at the end of each run. Instead, we enable open-close versioning for each file. As shown in Table 3.2, the copy-based physical layer suffers due to the copying overhead, as we explained earlier.

Benchmark	No versioning	Copy-based	Copy-on-write-based
500 1-MB files	7.30	21.26	10.53
1 500-MB file	7.48	20.80	7.95

Table 3.2: Time in seconds for open-close versioning using copy-based and copy-on-write-based physical layer implementations.

The copy-on-write-based physical layer on the other hand incurs an overhead only when 500 1-MB files are are individually versioned. Versioning of a single 500 MB file does not exhibit any overhead. We examined this further, and we found individual flushing of metadata blocks to be responsible for this performance loss. As all files are open-close versioned, every file has an associated version volume. Each version volume's volume index file contains logical configuration entries that track file history. Thus, for 500 files, there exist 500 version volumes, each having a volume index file containing one data block with logical configuration entries. In our current implementation, these 500 blocks are flushed using individual write operations. This results in multiple, small, random writes at the disk, and the resulting seeks result in performance loss. We are working on fixing this problem by vectoring these write requests in a single write operation using the vwrite call we introduced to solve a similar problem with small files [9].

3.8.4 Overhead of file volume virtualization

We now evaluate the overhead of our new infrastructure using two macro-benchmarks: (1) PostMark, configured to perform 20,000 transactions on 5,000 files, spread over 10 subdirectories, with file sizes ranging from 4-KB to 1-MB, and read/write

granularities of 4-KB, and (2) an application-level macro-benchmark, which we will refer to henceforth as *Applevel*, which consists a set of very common file system operations including copying (a complete MINIX 3 source tree), compiling (running "make clean world"), and running find and grep (searching for a keyword in all source and header files).

Benchmark	**Loris-V1 (w/o virtualization)**	**New Loris**
Postmark	686.00	693.00
Applevel (copy)	124.00	134.00
Applevel (build)	112.00	113.00
Applevel (find and grep)	20.00	19.00

Table 3.3: Transaction time in seconds for Postmark and wall clock time in seconds for Applevel tests

Table 3.3 shows PostMark and Applevel results for both Loris-V1 and our latest Loris version that supports all the new functionalities described in the paper. As can be seen, file volume virtualization in Loris has very little overhead, if any. Most other systems maintain elaborate block mapping information to virtualize file volumes, and hence suffer from performance degradation due to increased metadata footprint. As no such mapping information is maintained by Loris, there is no performance impact.

Benchmark	**No Snapshot**	**Copy-based**	**Copy-on-write-based**
Applevel (build)	123.00	131.00	124.00
Applevel (find and grep)	21.53	38.68	21.60

Table 3.4: Wall clock time in seconds for applevel tests using copy-based and copy-on-write-based physical layer implementations.

Table 3.4 shows an interesting comparison of the two snapshoting approaches using the Applevel benchmark. To perform this evaluation, we modified our test suite to take a file volume snapshot after the copy phase. As can be seen copy-on-write snapshoting does not suffer from any overhead in both build and find phases.

Copy-based snapshoting on the other hand suffers from a small overhead during the build operation, and a huge overhead during the find and grep operation. The find and grep operations result in the access time of all source and header files being updated. Since the access time is stored as an attribute by the physical layer, setting a new access time is done by making a setattr call. This call triggers a file snapshot operation at the logical layer. The copy-based physical layer copies over

the entire file data to create a new current version, while the copy-on-write physical layer just allocates a new inode and marks data blocks as shared. This is the reason behind the poor performance of the copy-based physical layer. Turning off access time updates resulted in similar performance figures for both copy-based and copy-on-write physical layer implementations.

We are working on a new naming layer that provides structured data storage to applications. The new naming layer will provide a new directory storage and indexing scheme. It will also be responsible for storing attributes with directory entries, and providing snapshoting of attributes. With this new naming layer, the physical layer will be in charge of snapshoting only file data, and thus the copying overhead will not be incurred for attribute changes.

3.9 Comparison with other approaches

Several commercial and academic projects have taken other approaches toward virtualizing file volumes. We will first discuss device management alternatives, and compare file pools with other approaches. Then, we will discuss file volume virtualization, and present the advantages that Loris has over other techniques. We discussed logical volume managers in great detail earlier in this paper. Most block-level virtualization solutions suffer from problems similar to the ones mentioned in Sec.3.2, and so we will not discuss them in further detail here.

3.9.1 Device management

Sun's ZFS [6] proposed refactoring the traditional storage stack to solve many problems we presented earlier. ZFS introduced storage pools, a new storage model for simplifying device management. Storage pools are based on the idea that block allocation decision is made by the wrong layer in the traditional stack—the file system layer. In the ZFS stack, a separate storage pool allocation (SPA) layer manages a pool of storage devices, and provides an interface for allocating and freeing virtual blocks, similar to the malloc() and free() interface for virtual memory. As the SPA provides a virtualized block address space, multiple file volumes can share a single storage pool. The SPA also simplifies and automates addition and removal of devices.

Our approach (file pools) offers advantages in addition to the benefits offered by storage pools. Providing RAID and volume management services at a file-level makes it possible to support advanced functionalities, like snapshoting, at several granularities, thus improving flexibility. Rather than bundling allocation and storage management together like ZFS, Loris makes a clean split between the two functionalities, by assigning block allocation to the physical layer, and storage management to the logical layer. The improved modularity makes it possible to

support heterogeneous device configurations using custom layout designs without affecting RAID and volume management algorithms.

3.9.2 File management and file volume virtualization

AFS[85] was one of the first projects to promote the use of file volumes as administrative units. AFS was a client-server system, and AFS clients accessed files using a <volume identifier, file identifier > pair similar to Loris. The volume identifier was used to locate the server housing the file volume. On the server side, many volumes share a single disk partition, and administrators could associate usage quotas with file volumes. File volumes were supported by modifying the 4.2BSD on-disk file system, to include per-volume inode tables that translated the <volume identifier, file identifier> pair to an inode. Snapshoting was a nightly operation that was implemented by incrementing the link count on all inodes associated with a file volume, and block-level data sharing was not supported. In contrast, file volume virtualization Loris is layout independent, snapshoting is a flexible and instantaneous operation, and if required, a copy-on-write based physical module can be used to support fine grained block-level data sharing.

There are several file systems, both on-disk and stackable ones, that support snapshoting and versioning. Most versioning file systems, like ElephantFS [81], support only open-close versioning and do not support file system snapshots. File systems that do support both, like ext3cow [74] are on-disk file systems, and do not support virtualized file volumes. Stackable file systems like RAIF [49], and VersionFS [68] are extremely flexible, portable, and support open-close versioning and RAID algorithms on a per-file basis. However, unlike on-disk file systems, they suffer from performance problems due to double buffering, and data copying. Loris, on the other hand, implements portable file volume virtualization in the logical layer by relegating storage-efficient block-level data sharing to the physical layer. It supports flexible snapshoting and versioning with its policy-mechanism split. Thus, Loris has the advantages of all these systems with its modular division of labor without the disadvantages.

Flexol [26] is the file volume virtualization system from NetApp. The basic idea adopted by FlexVol is to virtualize file volumes by creating a file volume inside a file, in a lower file system. The recursive use of the WAFL file system provided file awareness to the virtualization layer, making it possible to support several advanced functionalities like snapshoting and cloning. However, the dual mapping information that needs to be maintained by FlexVol causes some performance degradation. Unlike FlexVol, Loris approach does not suffer from overhead due to metadata footprint as seen earlier. Furthermore, while FlexVol supports file volume snapshoting and cloning, it does not support individual file snapshoting or open-close versioning.

3.10 Future work

The design of file volume virtualization in Loris opens up several areas of future work. We will now discuss two main avenues of ongoing research.

3.10.1 Flexible cloning in Loris

As we mentioned in the previous section, a number of commercial projects have introduced file volume cloning in addition to snapshoting. We are currently working on adding support for flexible copy-on-write-based cloning at both per-file and file volume granularities. The mechanism that supports snapshoting can also be used to support cloning with minor modifications. A new *clone* operation will be added to the standardized file interface. When the logical layer receives a request to clone a volume, it first picks a new volume identifier, and allocates a new metadata entry in the volume index file. It then instantiates a *clone volume*, that is a writable clone of the parent, by copying the target volume's metadata to this new entry. A new field in the snapshot volume's metadata links it with the new child volume. Following this, the clone volume's epoch number is set to one higher than its parent's epoch number. Individual files themselves are cloned on demand, just like snapshoting.

Copy-based snapshoting physical layer can support cloning without any modification, as each inode is an independent copy, not sharing any data blocks with other snapshots. However, supporting copy-on-write based cloning requires some changes to the physical layer to make sure that data blocks belonging to snapshot inodes are not freed while there are clones using those blocks.

3.10.2 Hybrid file pools

As we mentioned earlier when discussing file pools, several file allocation algorithms can be employed by the file pool sublayer to satisfy file creation requests. In addition, we are investigating the addition of device-aware migration algorithms at the file-pool sublayer. Since the file pool manages devices of multiple types, it can collect aggregate performance statistics of these devices. Based on these metrics, it can classify each device as belonging to a particular device type category. Similarly, since the file pool sublayer works at a file-level, it can also collect file access patterns on a per-file basis, using which, it can classify each file as belonging to a particular file type category. An easily configurable rule table can then be used to match file types with device types.

As an example configuration, small, read-only files could be positioned on a device with good random read performance, while small, write-mostly files could be positioned on a device with a log-structured layout. Large files that are both randomly read and written, on the other hand, could use two physical

modules, one with a log-structured layout for absorbing writes, and one on a device with good random read performance. The logical layer would direct writes to the write-optimized physical module, and migrate data in the background to the read-optimized device. We use the term *Hybrid file pools* to refer to this new storage model, as file pools can accommodate heterogeneous devices in hybrid configurations.

3.11 Conclusion

Virtualizing file volumes makes it possible to retain administrative flexibility without sacrificing storage efficiency. In this paper, we examined the traditional approach of virtualizing file volumes along two dimensions, namely, flexibility, and heterogeneity. We illustrated several problems associated with the traditional approach of virtualizing file volumes. We then presented our file volume virtualization design based on Loris, our fresh redesign of the traditional storage stack. We showed how Loris, with its unified, modular infrastructure, supports file volume snapshoting, per-file snapshoting, and open-close versioning.

Chapter 4

Efficient, Modular Metadata Management with Loris

Abstract

With the amount of data increasing at an alarming rate, domain-specific user-level metadata management systems have emerged in several application areas to compensate for the shortcomings of file systems. Such systems provide domain-specific storage formats for performance-optimized metadata storage, search-based access interfaces for enabling declarative queries, and type-specific indexing structures for performing scalable search over metadata. In this paper, we highlight several issues that plague these user-level systems. We then show how integrating metadata management into the Loris stack solves all these problems by design. In doing so, we show how the Loris stack provides a modular framework for implementing domain-specific solutions by presenting the design of our own Loris-based metadata management system that provides 1) LSM-tree-based metadata storage, 2) an indexing infrastructure that uses LSM-trees for maintaining real-time attribute indices, and 3) scalable metadata querying using an attribute-based query language.

4.1 Introduction

For over four decades, file systems have treated files as a set of attributes associated with an opaque sequence of bytes, and have provided a simple hierarchical structure for organizing the files. By providing a thin veneer over devices, and by not imposing any structure on the data they store, file systems have found widespread adoption in many application areas as preferred lightweight data stores. However, this very same generality has also led to the emergence of domain-specific, user-level metadata management systems in each application area to offset several shortcomings of file systems.

In the personal computing front, file systems have been used as document stores for housing a heterogeneous mix of data ranging from small text files to large multimedia files like photos, music and videos. With the amount of data stored by users increasing at an alarming rate, hierarchy-based file access and organization has lost ground to content-based access mechanisms. Most users have resorted to using attribute-based or tag-based naming schemes offered by multimedia and desktop search applications for managing and searching their data. These applications essentially build a user-level metadata management system that crawls the file system periodically to extract metadata, maintains indices on the extracted metadata, and offers application-specific search interfaces to query over metadata.

Modern-day enterprise storage systems house millions of files, and as each file has at least a dozen attributes (POSIX and extended attributes), these systems store an enormous amount of metadata. In addition, storage retention requirements for meeting regulatory compliance standards further fuels metadata growth. Administrators of such systems are constantly faced with the necessity to answer questions about file properties to make policy decisions like "which files can be moved to secondary storage?" or "which are the top *N* largest files?." Answering these questions require searching for relevant attributes over massive amounts of metadata. As using primitive utilities like the UNIX *find* utility at such large scales is not an option, administrators resort to using enterprise search tools. These applications build a user-level metadata management subsystem that gathers metadata periodically, maintains elaborate indices to speed up metadata queries, and offers administrator-friendly search interfaces.

In high-performance scientific computing, local file systems have been used as data stores in local nodes for multi-node, POSIX-compliant cluster/parallel file systems. Similar to enterprise systems, these systems also suffer from problems of scale. In addition, data provenance has emerged to be an extremely important technique in scientific computing for assessing the accuracy and currency of data. Several systems have proposed integrating provenance with parallel file systems to ensure complete, automatic provenance collection [15]. Such systems provide a metadata management system on top of local file systems that collect and store

provenance using optimized storage formats, index provenance records, and support specialized query languages for querying provenance data.

The data-intensive scalable computing front has witnessed the growth of domain-specific distributed file systems [30]. These systems maintain separate data and metadata paths so that after a single-step authentication at the metadata server, clients can retrieve data directly from data nodes, thus preventing any single data server from becoming bottlenecked. While such architectures have scaled the data wall, they continue to remain bottlenecked when it comes to metadata scalability. Scaling directory operations to support millions of mutations and lookups per second is an ongoing topic of research. Recent research has shown how indexing algorithms employed by local file systems can have a profound impact on performance of distributed directory partitioning and indexing schemes [72]. Some researchers have even proposed using custom-built databases that use sophisticated indexing structures to optimize metadata storage and retrieval, as storage back-ends for metadata servers [92]. These databases act as dedicated metadata management systems that obviate the need for using local file systems to store directory and other file metadata.

Thus, domain-specific metadata management systems have emerged as the "least common denominator" functionality across these application areas. However, such systems suffer from several problems that are well known [59]. First, since they are situated outside the mainline metadata modification path, they do not maintain indices in real time, which can result in stale query results due to inconsistent metadata. While this situation can be averted by updating indices frequently, user-level systems avoid this to reduce the performance impact caused by file system crawling. Unoptimized metadata placement makes crawling for metadata gathering an extremely slow, resource intensive operation. To avoid crawling the entire file system, some user-level systems [60] leverage new file system functionality like snapshoting and perform an incremental scan of only data modified since the last snapshot. While this in combination with other notification-based techniques can reduce the performance impact of crawling, it hardly helps to remedy the storage inefficiency inherent to user-level metadata systems. This inefficiency arises because metadata is stored twice, once in the file system itself, and a secondary copy in the elaborate indices maintained by user-level metadata systems. As metadata can consume a significant percentage of the storage capacity in large installations [59], this metadata duplication results in inefficiency that is unwarranted since the duplicated metadata is usually inconsistent.

Thus, once the purview of local file systems, metadata management is now being performed by domain-specific, user-level systems that suffer from several shortcomings. Rather than building custom databases for storing metadata, we believe that the right solution to these problems is integrating support for metadata management into local file systems. First, since file systems are in the mainline path of all metadata operations, no separate polling or notification mechanism is

needed for collecting changes. Second, since file systems are in charge of storing metadata, they can employ sophisticated, search-friendly storage formats to optimize metadata performance. Third, file systems can unify metadata storage and indexing using a single storage format, thus avoiding unnecessary duplication. Fourth, with metadata management being the least common denominator functionality, it is obvious that an integrated system can be used as a customizable framework for deploying domain-specific solutions. Such a customizable framework should possess two salient properties. First, it should be efficient; the integration of metadata management should not cause performance deterioration. Second, it should be modular; the metadata management functionality should be independent of other functionalities. Unfortunately, traditional file systems lack the latter property.

Traditional file systems use customized data structures for storing metadata and such data structures form an integral part of the file system's on-disk layout. Further, file systems handle a range of tasks from providing device-specific layout algorithms to implementing POSIX-style file and directory naming. As a result, any metadata management system that is integrated with one file system is inherently non-portable to other file systems, and a single implementation of metadata management cannot be used across multiple file systems. Thus, metadata management would have to be implemented on a case-by-case basis due to the lack of modularity of traditional file systems.

In prior work [9], we presented Loris, a complete redesign of the traditional storage stack that solves several reliability, flexibility, and heterogeneity issues by design. In this paper, we show how Loris can be used as an efficient, modular metadata management framework. In doing so, we present our Loris-based metadata management system that provides 1) LSM-tree-based metadata storage, 2) an indexing infrastructure that uses LSM-trees for maintaining real-time attribute indices, and 3) scalable metadata querying using an attribute-based query language.

The rest of the paper is organized as follows. In Section 4.2, we present an overview of Loris and show how the Loris stack provides a convenient framework for integrating metadata management. We present the design of our metadata management system in detail in Section 4.3, following which we present an evaluation of our prototype using several micro and macrobenchmarks in Section 4.4. We then compare our approach with related work in Section 4.5. Finally, we present future work in Section 4.6, and conclude in Section 4.7.

4.2 Background: The Loris Storage Stack

In prior work [8], we highlighted a number of fundamental reliability, flexibility, and heterogeneity problems that plague the traditional storage stack and we presented Loris, a fresh redesign of the stack, showing how the right division of labor

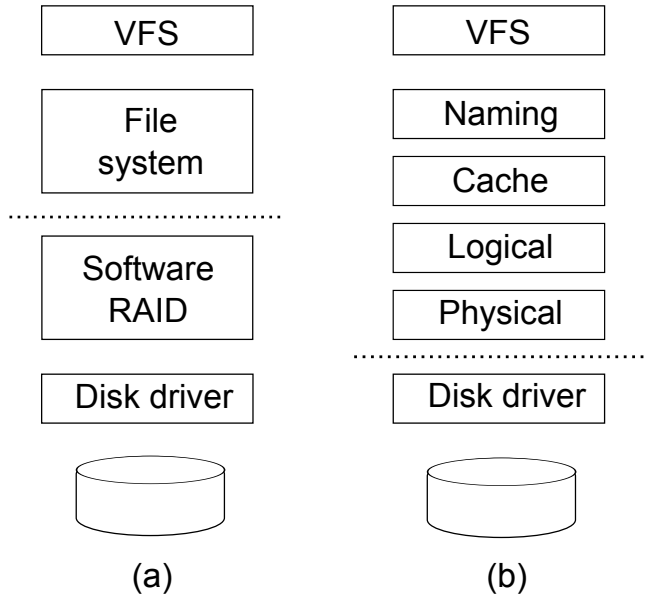


Figure 4.1: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file-aware; the layers below are not.

among layers solves all problems by design [9]. In this section, we will briefly describe Loris' layered architecture.

Loris is made up of four layers, as shown in Figure 4.1. All the layers in Loris are file-aware, in contrast to the traditional stack, and the interface between the layers is file-centric, consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. Files are identified throughout the stack using a unique *file identifier* (file ID). Each Loris file is also associated with several *attributes*, and the interface supports two attribute manipulation operations—*getattribute* and *setattribute*. We will now briefly describe the responsibilities of each layer in a bottom-up fashion.

4.2.1 Physical layer

The physical layer exports a "physical file" abstraction to the logical layer. The logical layer stores data from both end-user applications, and other Loris layers in physical files. The physical layer is tasked with two primary responsibilities. The first responsibility of the physical layer is providing persistent storage of files and their attributes using device-specific layout schemes. Each storage device is managed by a separate instance of the physical layer, and we call each instance a *physical module*. Each device, and hence its physical module, is uniquely identified using a *physical module identifier*. Our prototype physical layer was based on

the traditional UNIX layout scheme. The physical file abstraction is realized using inodes. The logical layer uses an inode number to refer to the target physical file in any file operation.

The second responsibility of this layer is providing data verification. Each physical layer implementation must support some comprehensive corruption detection technique, and use it to verify both data and metadata. As the physical layer is the lowest layer in the stack, it verifies requests from both applications and other Loris layers alike, thereby acting as a single point of data verification. Our prototype physical layer supports parental checksumming [55] and uses it to provide end-to-end data integrity.

4.2.2 Logical layer

The logical layer is responsible for providing per-file RAID services using physical files. The logical layer multiplexes data across multiple physical files and provides a virtualized *logical file* abstraction. A logical file appears to be a single, flat file to the cache layer above it. The logical layer abstracts away details like the physical files that constitute a logical file, the RAID level used by a file, etc. by using the logical file abstraction.

The central data structure in the logical layer that enables multiplexing is the *mapping file*. This file contains an array of *configuration information* entries, one per logical file. Each configuration information contains: (1) the RAID level used for this file, (2) the stripe size if applicable and (3) <physical module identifier, inode number> pairs that identify the physical files that make up this logical file. Since the mapping file is an extremely crucial piece of metadata, it is mirrored on all physical modules. A physical file with a fixed inode number is reserved in each physical module and is used to store the mapping file's data blocks.

Let us consider a logical file with file ID F1, that is stored using a RAID 0 configuration backed by physical files with inodes I1, I2 on physical modules P1, P2 respectively. Such a file would have F1=<raidlevel=0, stripesize=4096, <PF1=<P1:I1>, PF2=<P2:I2>> as its configuration information in its mapping entry. We will explain the entry creation process later while describing the naming layer. For now, let us consider a read request for this logical file. When the logical layer receives a request to read, say, 8192 bytes, from offset 0, it determines that the logical byte range 0-4095 maps onto the byte range 0-4095 in physical file PF1 and the logical byte range 4095-8191 maps onto the range 0-4095 in physical file PF2. Having determined this, the logical layer forwards a request to read 4096 bytes, at offset 0, from files PF1 and PF2 to physical modules P1 and P2 respectively.

4.2.3 Cache layer

The cache layer provides data caching on a per-file basis. As it is file-aware, it can deploy different data staging and eviction policies for different files depending on their types and access patterns. Our prototype cache layer provides a simple fixed-block read ahead and LRU-based eviction for all files.

4.2.4 Naming layer

The naming layer acts as the interface layer. Our original prototype naming layer provided POSIX-style file/directory naming and attribute handling. The naming layer is also responsible for assigning a unique file ID to each Loris file. It processes a file create request by picking a unique file ID and forwarding the create call to the cache layer, which, in turn, forwards it to the logical layer. The logical layer picks physical modules for this logical file, and forwards a create call to those modules. The physical modules service the create call by allocating physical files and returning back their inode numbers. The logical layer records the <physical module identifier, inode number> pairs, in addition to other details, in the mapping file's configuration entry.

Directories are implemented as files containing a list of records that map file names to Loris file IDs. Only the naming layer is aware of this structure; below the naming layer, a directory is simply considered another opaque file. Loris attributes are used by the naming layer to store per-file POSIX attributes like modification time and access permissions. These attributes are passed from the naming layer to the corresponding physical modules using the setattribute call. Our physical layer implementation stores attributes in the corresponding physical file's inode. The getattribute call is used by the naming layer to retrieve the stored attributes when necessary. Loris attributes are also used to exchange policy information between layers. An example of this usage is how we enable selective mirroring of directories on all physical modules to improve reliability and availability. When the naming layer issues a create call for creating a directory, it informs the logical layer that the corresponding logical file must be mirrored by passing the RAID level (RAID 1 on all physical modules) as an attribute. This attribute is not passed down for normal files. Thus, the attribute infrastructure in Loris serves a dual purpose.

4.3 Efficient, Modular Metadata management with Loris

With the modular division of labor between layers in the Loris stack, the naming layer in Loris provides an ideal place for integrating metadata management. Since all layers in the Loris stack are file-aware, the cache, logical and physical layers can be conceptually seen as providing a file store for the naming layer. The

naming layer could thus implement custom storage schemes that pack domain-specific metadata into performance-optimized file formats that would be stored by the lower layers as plain Loris files. By isolating metadata management in the naming layer, Loris provides a modular framework where naming implementations can be changed without affecting algorithms in other layers.

In this section, we will detail the design of our new naming layer that provides metadata management. It is made up of two sublayers, namely, the *storage management sublayer* and the *interface management sublayer*.

4.3.1 Storage management sublayer

The storage management sublayer is the lower layer and is responsible for providing domain-specific storage formats that optimize storage and retrieval of metadata. It provides a simple key-value interface to the interface management sublayer, and performs space-efficient packing of such key-value pairs in Loris files.

Our storage management sublayer uses write-optimized log-structured merge (LSM) trees [70] for storing key-value pairs. LSM-trees are multi-component data structures that consist of a number of in-memory and on-disk tree-like components. The fundamental idea behind maintaining two different types of components is to buffer updates temporarily in the in-memory component and periodically flush out batched updates as new on-disk components. These on-disk components are write-optimized tree structures that provide space-efficient packing of key-value pairs by filling up tree nodes completely. They are immutable, and thus, once created, they can either be deleted as a whole, or be used for key lookups, but can never be updated in place. By buffering updates in memory and writing them out in batches to a new on-disk component, LSM-trees avoid directly updating on-disk components, and thus eliminate expensive seek operations.

A lookup operation in an LSM-tree first checks the in-memory tree for the target key. A failure to locate the key results in searching the on-disk components chronologically. By using components that are tree structured, LSM-trees provide efficient indexing of data in both in-memory and on-disk components. However, the number of on-disk components that must be searched plays a crucial role in determining the overhead of lookup operations. Minimizing this overhead requires periodic merging of on-disk components to form a single densely-packed index. Because on-disk components are immutable, such a merge operation can happen asynchronously, in the context of a background thread without affecting foreground traffic.

There are two special boundary cases that arise when one uses an LSM-tree-based key-value store. The first one concerns updates to existing records. Updating an existing record is performed by adding a new record to the tree with the same key and the updated value. Lookups are executed by chronologically searching all components and returning as soon as there is a match, so new records im-

plicitly obsolete any other records that exist in the tree with the same key. The second boundary case concerns record deletion. Delete operations are performed by inserting "tombstone" markers—records whose value denotes that the key-value pair has been deleted. If a key lookup operation ends up at a tombstone record, the lookup fails with an error that notifies the caller that the record being looked up has been deleted. In both boundary cases, the old, outdated records consume unnecessary space and are cleaned periodically during merging to improve space utilization.

We will explain how the LSM-tree-based key-value store is used to house POSIX metadata in Sec. 4.3.2, but for now, we would like to emphasize that LSM-trees are ideally suited for storing metadata due to several reasons. First, metadata updates are rarely sequential. Most modern file systems use B-tree variants for storing metadata. It is well known that B-trees (and their variants) require random writes for random updates and may get fragmented over time [92]. Further, almost all existing storage technologies are ill-suited for such random-write workloads. Despite tremendous growth in capacity and bandwidth of disk drives, the performance of small, random workloads continues to suffer from seek-imposed access latencies. RAID installations using parity-based redundancy schemes have known issues with small, random-write workloads [93]. Even modern SSDs suffer under a random-write workload and it has been shown that random writes can significantly reduce both the performance and lifetime of SSDs [75]. By using write-optimized LSM-trees, we convert slow, small, random metadata updates into fast, large, sequential write operations without sacrificing lookup performance.

Second, metadata lookups exhibit significant locality. A directory listing operation for instance looks up POSIX attributes of all files in a directory. A backup application might scan through extended attributes of each file in the file system to identify flagged files that must be incrementally backed up. Most file systems fail to exploit such locality as metadata is scattered all over the device. For instance, while file names are stored in directory data blocks, POSIX attributes are stored in inodes, and extended attributes are stored in blocks pointed to by inodes. As a result, metadata lookups result in expensive seek operations when disk drives are used, significantly crippling performance. In contrast, with LSM-trees, locality can be controlled with the choice of key format. As records in the leaves of both the in-memory tree and on-disk trees are sorted in key-order, iterating over records with the same key prefix is very efficient as they are more likely to be stored together on disk, and thus more likely to reside in the (block-level) cache. We will see later how we take advantage of this to achieve good directory listing performance.

Third, a significant number of files in several workloads exhibit very short lifetime. For instance, it has been shown that about 50% of files are deleted within 5 minutes, with 20% existing for less than half a minute in certain local file system workloads [71]. Development workloads also create a large number of empty lock

files, and short-lived compiler temporaries. As we will see later, supporting native searching in our naming layer requires the capability to create a large number of links to existing files, which may be temporary when queries are used for a one-time, dynamic search over metadata. Traditional file systems typically require additional implementation tricks to optimize for such short-lived metadata. As metadata updates are first handled in memory, LSM-trees handle such temporary files efficiently.

Our implementation employs AVL-trees for the in-memory component (but any search-efficient data structure will suffice), and densely-packed, two-level B^+-trees for storing on-disk components. As the on-disk components are immutable, nodes are packed full for optimal space efficiency. To limit the number of disk seeks to one per disk component, their root nodes are always kept in memory. Merge parameters, like component size thresholds, maximum number of disk components, etc., are configurable to allow system-specific optimization.

In addition to the record-based lookup/insert operations, our LSM-storage sublayer's interface also exposes a *prefix lookup* operation that returns an iterator over all records whose key starts with the supplied prefix. We mentioned earlier that records with the same key prefix are likely to be stored together on disk; enumerating them can therefore be performed efficiently. Thus, while the choice of key format is used to control locality, the prefix lookup can be used as a means to exploit it.

4.3.2 Interface management sublayer

The interface management sublayer is responsible for translating domain-specific interface operations to key-value insertion or lookup operations on the underlying storage management sublayer. We will first explain how this sublayer maps well-known POSIX abstractions to key-value pairs, thereby providing POSIX-compatible naming. Then, we will describe extensions that provide scalable attribute-based metadata search using LSM-tree-based attribute indices.

POSIX interface

The interface management subsystem provides the POSIX environment to applications by mapping familiar file system primitives to key-value pairs. Thus, while the storage management subsystem stores key-value pairs, the interface subsystem controls the semantics of keys and values. We will now describe how we map per-file POSIX attributes and directories to key-value pairs.

As we mentioned earlier, each file in Loris is identified using a unique file ID. As each file in POSIX is associated with a set of POSIX attributes, a straightforward way to map these attributes would be to use the unique file ID as the key, and store all attributes as the value. It is important to note here that only file attributes

are stored by the storage subsystem, not file data. Thus, a file create request would result in the interface management subsystem storing a new <file ID, <POSIX attributes>> pair in the LSM-tree, following which, the create call would be forwarded to the lower layers. While subsequent metadata updates would result in the LSM-tree being updated, data updates would be immediately directed to the lower layers.

Supporting directories is a bit tricky. With our old naming layer, directories were files containing an array of <name, file ID> pairs, one per file stored in that directory. Since each directory is also a file, it also has its own unique file ID. Thus, one possible implementation would be to use the directory's file ID as the key and store this array of entries as the value. However, directories also have POSIX attributes associated with them, and hence, each directory could then be represented using two key-value pairs, one containing an array of entries, and the other containing the POSIX attributes of the directory itself. While this approach is simple to implement, it however suffers from the disadvantage that a lookup operation has to perform a linear scan through file names.

Avoiding such a linear scan requires using LSM-trees to index directory entries. Such an index would use file names as keys to lookup file identifiers. However, the key structure of such a tree is incompatible with the key structure for storing file attributes we mentioned earlier. Thus, using this approach requires maintaining two LSM-trees, one mapping file names to file identifiers, and the other mapping file identifiers to file attributes. The level of indirection also means that a directory listing operation would need to perform two lookups, one per tree. While the resulting implementation would be better than existing file systems due to the use of LSM-trees to store POSIX attributes, we wanted to eliminate this additional indirection to maximize performance gains.

Eliminating this level of indirection requires using a single tree that uses a unified key structure. Thus, we adopted an approach similar to the one used by BabuDB [92] for mapping both directories and POSIX attributes to key-value pairs. In this approach, we use the triplet <parent directory's file ID, file name, metadata type> as the key to store POSIX attributes for each file. The reasoning behind using this triplet is to speed up lookup operations. Each lookup operation attempts to resolve a filename in the context of a directory to retrieve the target file's identifier or file attributes. Since each directory in Loris is a file, and hence has a file ID, using the directory's file ID in combination with the file name can be used to search through the index for the relevant file's attributes.

The metadata type field in the key is an optimization to reduce the amount of metadata updates. It essentially classifies metadata into two categories: frequently-updated metadata (like access time and size), and rarely-updated metadata (such as modes, ACLs). By making this classification, updates to the LSM-tree are kept small due to the fact that a change to the frequently-updated metadata does not involve writing out all POSIX metadata. Thus, each file is associated with two

key-value pairs, one per metadata type, and both these pairs can be located using the key-prefix <parent directory's file ID, file name>.

With this metadata scheme, we achieve several advantages when compared to our old naming layer. First, by linking files and directories in reverse with the parent's file ID, directories no longer need to store <file name, file ID> pairs in data blocks. As a result, directories are empty files represented in the LSM-tree using two key-value pairs that record the directory's POSIX attributes. Thus, unlike our old naming layer, directory create requests need not propagate down to the lower layers, thus improving performance, as it eliminates further processing of create requests by these lower layers. For similar reasons, metadata updates are also significantly faster. As the old naming layer used Loris' attribute infrastructure to store POSIX attributes, for each change in any POSIX attribute, the naming layer had to use a setattribute call that percolated down the stack resulting in unnecessary overhead. The new naming layer completely avoids this as metadata updates are restricted to the LSM-tree.

Second, lookup operations are much more scalable as linear lookups are avoided in both in-memory and on-disk trees. Third, by using parent ID as a part of the key, lookups can benefit from significant locality. This is due to the fact that records in the leaf of the tree are sorted based on their parent ID and then by their file name. Thus, all file entries belonging to a directory are tightly packed, thereby speeding up operations like directory listing. Fourth, using LSM-based storage results in random metadata updates being converted into sequential write operations at the storage level, improving performance significantly. Our old naming, in contrast, incurred expensive disk seeks for each metadata update as POSIX attributes were stored in the inodes. Table 4.1 illustrates the mapping of metadata to key-value pairs using an example configuration.

Linking files to their parent however does complicate the implementation of hard links, as a file's metadata is mapped to one unique <parent ID, filename> pair. Hard links require a file to be accessed from multiple names—this requires either storing the metadata redundantly for each name, or letting each name point to one central entry. The first approach will not scale as the number of links to a file increases, while locality advantages are lost in the second approach. In the latter case, this means one extra index traversal for each hard linked file. As hard links are not very common, we believe such an overhead is acceptable, and we do not duplicate metadata.

Our implementation stores the metadata of hard linked files under the different key <hardlink ID, file ID, type> (where hardlink ID is simply a reserved parent ID). Thus, metadata is retrieved in the same way, except by file ID rather than parent ID and filename. The original <parent ID, filename, type> record is still used, but only includes the file ID and a flag denoting that the name represents a hard linked file. As a result, enumerating the names in a directory can still be done with a single prefix lookup, but stat lookups require extra index traversals.

Key	Value
<0, /, f>	atime=2011-01-01 . . .
<0, /, r>	id=1 links=4 mode=drwxr-xr-x . . .
<1, etc, f>	atime=2011-01-02 . . .
<1, etc, r>	id=5 links=2 mode=drwxr-xr-x . . .
<1, tmp, f>	atime=2011-01-03 . . .
<1, tmp, r>	id=3 links=2 mode=drwxr-xr-x . . .
<3, prog.c, f>	atime=2011-01-01 . . . size=2000
<3, prog.c, r>	id=10 links=1 mode=-rw-r–r– . . .
<3, test.txt, f>	atime=2011-01-03 . . . size=100
<3, test.txt, r>	id=13 links=1 mode=-rw——- . . .
<5, passwd, f>	atime=2011-01-02 . . . size=1024
<5, passwd, r>	id=20 links=1 mode=-rwx—— . . .

Table 4.1: Example mapping of POSIX metadata to key-value records. This example illustrates a small file tree containing the root directory /, the directories /etc and /tmp, and the files /etc/passwd, /tmp/test.txt and /tmp/prog.c. Keys are in the format <parent ID, filename, type>, causing metadata records of files in the same directory to be stored adjacently. Record values contain either frequently-updated metadata when the type value is 'f' and rarely-updated metadata when the type value 'r'. Metadata is shown here in human-readable format but is physically stored in native format using positional notation.

Efficient metadata search

Having explained how the conventional POSIX interface can be realized using our infrastructure, we will now describe extensions to the interface management subsystem that enable metadata search. Since metadata is well structured and can be logically considered to be a collection of attribute-value pairs associated with files, most user-level metadata management systems have adopted an attribute-based scheme for naming and searching. We will now show how our interface management sublayer can be extended to support such an attribute-based naming scheme. While we use attribute-based naming as an example, we would like to point out here that the infrastructure is flexible enough to accomodate other naming schemes, like tag- or keyword-based schemes, as well.

Real-time indexing The first requirement for enabling efficient searching is attribute indexing. Consider a name lookup operation. Such an operation can be considered to be a search over POSIX metadata for the name attribute. In the absence of indexing, one would have to resort to a linear lookup over all file names, similar to our earlier prototype. As we mentioned earlier, we solved this one specific problem by indexing file names. However, such an index is not usable for

Key	Value
<atime, 2011-01-01, 20>	
<atime, 2011-01-02, 13>	
<atime, 2011-01-03, 10>	
⋮	⋮
<size, 100, 13>	
<size, 1024, 20>	
<size, 2000, 10>	
⋮	⋮

Table 4.2: Attribute index belonging to the example in Table 4.1. Only files are indexed. This shows the subset of the index that covers the *atime* and *size* attributes. The value fields are unused.

searching over any other attribute. For instance, a search for files with size greater than 1 GB can be done only by performing a linear scan over each file's metadata. Though the locality-friendly, densely-packed LSM-tree design makes high-speed sequential scans possible, this approach does not scale as large installations can have millions of files.

To avoid linear lookup, we index attributes in an auxiliary LSM-tree that maps attribute values to file IDs. This tree contains records with the <attribute ID, attribute value, file ID> triplet as the key, and an empty value field. The file ID needs to be part of the key in order to guarantee its uniqueness. Table 4.2 shows a sample index tree for the example configuration in Table 4.1.

Choosing which attributes to index is a policy decision that can be tuned for each installation and the ones that are indexed are identified in the tree using the attribute ID. Because LSM-trees are update-optimized, maintaining these indices is cheap. We further minimize their overhead by separating metadata trees and index trees. This allows us to adopt different merge parameters for index and metadata trees. For instance, since metadata search is relatively less frequent compared to index updates, we could trade off query performance for improving indexing performance by maintaining a higher number of on-disk components to store indexing entries, thereby delaying expensive merge operations.

Attribute-based search interface Having described the indexing mechanism, we will now detail attribute-based query processing. Exposing new interfaces and functionalities can generally be done in two ways: 1) by extending APIs and system calls by introducing new function calls that applications can directly invoke, or 2) by overloading the semantics of objects in an existing interface. The latter is preferable for integration and compatibility reasons [32]. In the case of

Key	Value
1	<0, />
3	<1, tmp>
5	<1, etc>
10	<3, prog.c>
13	<3, test.txt>
20	<5, passwd>

Table 4.3: Name index belonging to the example in Table 4.1. As an example, the fully qualified path name to file ID 20 is retrieved as follows: first, key "20" is looked up to get parent ID 5 and name passwd. Next, key "5" is looked up to get parent ID 1 and name etc. At this point, the path is etc/passwd. Next, 1 is looked up, and the root is reached. Thus, the full path is /etc/passwd.

file systems, the POSIX-based interface is ubiquitous, and our goal is to preserve backward compatibility as much as possible. Therefore, we overload POSIX semantics without changing the VFS interface.

In Loris, we generalized the concept of a virtual directory [32] to provide *typed virtual directories*. Just like the Semantic file system [32], directories are considered virtual when their file entries are created on the fly. Each virtual directory is associated with a type that determines the mechanism that populates the file entries. For example, a search virtual directory is a virtual directory whose directory listing implementation provides query resolution. A version virtual directory [10], however, has a directory listing implementation that enumerates all versions belonging to a particular file. In this paper, we will describe only our search virtual directory's mechanism.

In Loris, query results are exposed through search virtual directories that are instantiated dynamically using a *well-formed query term* specified by the user. A query term is a boolean combination of attributes and associated conditions that must be met for a file to be a part of a search virtual directory. Such a query starts with '[' and ends with ']'. Our current prototype supports equality conditions that can be used to perform a point search over specific attributes. For instance performing a directory listing of the search directory with the name "[uid=100]" results in all files owned by the user with uid 100 being listed. Similar to Semantic FS [32], search directories can also be used to map conjunctive queries into tree-structured path names. For example, performing a directory listing of the search directory "[uid=100]/[size>1048576]" results in all files owned by the user with uid 100 and having size over 1 MB being listed.

Our search directory implementation performs query resolution in two steps. In the first step, the attribute index is referenced to determine which file IDs match the given query. For instance, the query "[uid=100]" results in our implementation

performing a prefix lookup with the key <METADATA_ID(uid), 100> on the index LSM-tree. Thus, using a single range lookup, we can identify the file IDs of all the files that match a query. If the query is the conjunction of two subqueries, each subquery is performed separately and the intersection of file IDs is used.

In the second step, we need to derive the fully qualified path name of each file armed with the file ID. This is required because our mapping of POSIX attributes to key-value pairs uses <parent directory's file ID, file name, metadata type> as the key. Thus, it is not possible to use just the file ID to retrieve file metadata. To solve this problem, we added a new *name index* that maps file ID to <parent directory's file ID, file name>. Thus, each path can be generated by looking up the parent ID and file name corresponding to each file ID in the result and traversing the parent ID chain all the way to the root directory. Table 4.3 shows the name index belonging with the example configuration from Table 4.1 and illustrates how it can be used to form a full pathname from a file ID.

We are currently experimenting with different methods for exposing the results of a query. One such method is creating a symbolic link entry for each file, with the link name being the file name, plus a suffix if it is not unique. We currently perform the query on the fly during a lookup request, but a better approach would be to cache the results in a separate LSM-tree. Not only is caching better for performance, it would also simplify the implementation of search directories since a lot of code can be reused.

4.4 Evaluation

In this section, we will evaluate several performance aspects of our naming layer. We implemented the new prototype as a part of the Loris stack running on the MINIX 3 multiserver operating system [44]. We will first present our microbenchmark-based evaluation that compares the scalability of our implementation with the original Loris naming layer. Following this, we will present a comparison of metadata storage/retrieval performance the two naming layers using *Postmark* and *Applevel* macrobenchmarks, and a comparison of query performance using native Loris queries vs. using the *find* utility. Finally, we will present an evaluation of our attribute indexing implementation.

4.4.1 Test setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4 GB RAM, and four 500 GB 7200RPM Western Digital Caviar Blue SATA HDD (WD5000AAKS). We ran all tests on 8 GB test partitions at the beginning of the disks.

Microbenchmark	Loris (MFS)	Loris (new)
Create		
Tree A (4 dirs × 25,000 files)	153.5 (1.0)	43.58 (0.28)
Tree B (10 dirs × 10,000 files)	76.23 (1.0)	45.88 (0.60)
Tree C (100 dirs × 1000 files)	47.98 (1.0)	42.76 (0.89)
Find and Stat		
Tree A	79.3 (1.0)	6.86 (0.09)
Tree B	27.5 (1.0)	6.00 (0.17)
Tree C	9.21 (1.0)	5.71 (0.62)
Random Update	502.4 (1.0)	330.1 (0.66)

Table 4.4: Wall clock time for several microbenchmarks. All times are in seconds. The table shows both absolute and relative performance numbers, comparing our presented naming layer with our MFS-based one.

4.4.2 Microbenchmarks

We used two microbenchmarks to evaluate the update and lookup performance of the LSM-tree-based naming layer. Our first microbenchmark created 100,000 files spread across 4, 10 and 100 directories, following which, our second microbenchmark performed the equivalent of "find | xargs stat" at the root directory. Table 4.4 outlines the running times of these microbenchmarks with the old and the new naming layers. The results show that the old naming layer does not scale as the directory size increases, while the new naming layer's performance remains consistent. Performance gains achieved by the new naming layer under the create benchmark can be attributed primarily to the indexed lookup of file names. The find benchmark on the other hand also benefits from two other factors. First, as our POSIX-mapping associates file metadata directly with file names, we avoid the additional level of indirection inherent to the old naming layer (name to inode number, and inode number to attributes). Second, tight packing of metadata in our on-disk trees resulting in increased cache hits due to the locality inherent in metadata requests.

MINIX 3 does not have file systems that support tree-based directory indexing. As we explained earlier, a significant portion of the performance improvement in the last two micro-benchmarks can be attributed to the indexed lookup of file names rather than the write-optimized LSM infrastructure. Since we wanted to isolate the performance gains of using the write-optimized LSM-tree, we built a random metadata update microbenchmark. In this benchmark, we perform 200,000 metadata operations (chmod, chown, utime) on randomly-chosen files in a three-level directory tree of 200,000 files. The number of files per directory was deliberately restricted to 64 (the number of entries per directory block in the old

Macrobenchmark	Loris (MFS)	Loris (New)
PostMark	744 (1.0)	511 (0.69)
Applevel		
Copy	46.4 (1.0)	40.9 (0.88)
Build	95.0 (1.0)	92.0 (0.97)
Find	22.4 (1.0)	10.7 (0.48)
Delete	32.6 (1.0)	29.1 (0.89)

Table 4.5: Transaction time for PostMark and wall clock time for Applevel benchmarks. All times are in seconds. The table shows both absolute and relative performance numbers, comparing our new naming layer with the original one.

naming layer) to avoid the lookup bottleneck of the old naming layer. Thus, each lookup operation in the old naming layer has to retrieve only one directory block, and perform a linear scan over 64 entries. At such a small scale, linear lookups provide performance comparable to indexed lookups. As shown in table 4.4, this benchmark shows an improvement of about 34%, which can be attributed to the batched flushing of metadata performed by the write-optimized LSM-tree.

4.4.3 Macrobenchmarks

Metadata performance

We used two macrobenchmarks, namely Postmark and Applevel, to evaluate the overall performance of the new naming layer. We configured PostMark to perform 20,000 transactions on 10,000 files, spread over 100 subdirectories, with file sizes ranging from 200 to 400 KB, and read/write granularities of 4 KB. Our application-level benchmark consists of a set of very common file system operations, including copying, compiling, searching, and deleting. The copy-phase involves copying over 75,000 files, including the MINIX 3 source tree. This source tree is compiled in the build-phase. The find-phase traverses the resulting directory tree and stats each file. Finally, the delete-phase removes all files. The results are listed in Table 4.5.

The PostMark numbers show a performance increase of roughly 31%, demonstrating the effects of better metadata management when working with many small files. The application-level benchmark is much more data-oriented than the previous benchmarks. Consequently the results only show a moderate increase in performance, except in the find-component of the benchmark, which is completely metadata-oriented and twice as fast.

Query	Indexed search	Find
Query 1	0.33 (1.0)	8.24 (25.0)
Query 2	0.30 (1.0)	7.90 (26.3)

Table 4.6: Time for attribute-based searches to complete, in seconds, comparing Loris indexed search with the same query done using the Unix *find* utility.

Query performance

We will now present our evaluation of the metadata search functionality in Loris. We evaluated the total time taken to resolve two typical administrative queries: 1) find all files owned by uid 100 with size $>$ 1 MB, and 2) find all files modified in the last hour. The queries were run over a randomly-generated, three-level hierarchy containing 200,000 files. The query results encompass 1% of the total amount of files.

We are currently experimenting with different interface setups and do not yet have a fully working query resolver. Instead, we simulated queries by hardcoding index lookups. Thus, the performance numbers presented here reflect only index lookup times and do not include the overhead of other aspects of query processing, like parsing. The running times are listed in Table 4.6. For comparison, we ran the same query with *find*. We see that our indexing scheme achieves excellent performance—both the range scans for getting the matching file IDs and the pathname generation step are performed almost instantly.

4.4.4 Attribute indexing overhead

Finally, we reran the previous macrobenchmarks (PostMark, and Applevel without delete) with attribute indexing on for the following POSIX attributes: *size*, *uid*, *gid*, *atime*, *mtime* and *ctime*. Only files are indexed, not directories. We relaxed the merging parameters of the LSM-tree used for indexing purposes and included the running times in Table 4.7.

The results show that the overhead of indexing hovers between 4–19% for our tests. File creates are the most expensive, as exemplified by the copy-phase of the Applevel benchmark. This is because each new file adds an index entry for each indexed attribute plus an index entry in the name index. We believe this overhead is acceptable as it is possible to perform selective indexing of both attributes and files easily to reduce the overhead.

Benchmark	Indexing disabled	Indexing enabled
PostMark	511 (1.0)	572.0 (1.12)
Applevel		
Copy	40.9 (1.0)	48.7 (1.19)
Build	92.0 (1.0)	105.2 (1.14)
Find	10.7 (1.0)	11.1 (1.04)

Table 4.7: Attribute indexing overhead measurements. Time for PostMark and Applevel macrobenchmarks to complete, in seconds, with attribute indexing enabled/disabled.

4.5 Related Work

In this section, we will present related work and compare our Loris-based metadata management infrastructure with other approaches. We classify related work into three categories based on whether it deals with metadata storage management, metadata interface management, or both.

4.5.1 Storage management

Today, most file systems use B-trees or their variants for indexing directory entries. Unfortunately, while these do provide efficient keyed lookup, it is well known that they are slow for high-entropy inserts due to their in-place updating of records which requires one disk seek per tree level in the worst case [92].

Spyglass is a user-level search application that proposed using multidimensional indexing structures in combination with hierarchical partitioning to provide scalable metadata search. In effect, Spyglass builds a user-level metadata management subsystem and as a result suffers from problems inherent to such systems. We, on the other hand, propose a modular integration of such functionalities within the Loris storage stack. We would like to point out here that even though we used LSM-tree-based indexing, our framework is flexible enough to support other types of indices. As we will discuss later, we are working on modifying our prototype to support and evaluate different partitioning strategies to provide scalable searching. BabuDB is a custom-built database back-end intended to be used as a metadata server back-end for any distributed file system. BabuDB uses LSM-trees for storing file system metadata and their mapping of hierarchical file systems to database records is similar to our approach. They also exploit the snapshoting capabilities of the LSM-tree to support database snapshots. As BabuDB targets only the storage management aspect of metadata management systems, it does not provide real-time attribute indexing, or search-friendly interfaces for querying metadata.

4.5.2 Interface management

In order to cope with the hierarchical model's restrictions, file systems have been fitted with hard and symbolic links, allowing files to be referenced from multiple names. Unfortunately, such links are unidirectional. A problem, for example, is that when you delete the symbolic link target, you end up with a dead link. Gifford et al. introduced the concept of Semantic File Systems (SFS), providing associative access to files [32]. Virtual directories are used to list files matching a specified attribute-value pair; file type-specific transducers are used to extract this metadata from file contents. Although the path-based queries are syntactically POSIX-compatible, only conjunctive queries are possible. Others have extended upon these concepts [14] [83] [95].

HAC [35] exposes similar functionality through semantic directories: persistent directories associated with a query that are updated periodically. Unlike SFS's read-only virtual directories, semantic directories are tightly integrated into the hierarchy, and HAC allows adding and removing files from them. HAC has been architected such that it is possible to choose different mechanisms for associative access. For example, rather than the default full-text retrieval scheme, more sophisticated schemes can be used. All these systems focus primarily on the interface specification to enable content-based access and do not consider storage management issues.

4.5.3 End-to-end metadata management

Since databases have optimized storage formats for storing structured data, Inversion proposed using a relational database as the core file system structure [69]. By using several PostgreSQL tables to store file metadata and data, Inversion was able to extend transactional semantics of databases to file systems. Further, such a database could also be queried declaratively to search over metadata. However, it has been shown that such a database-based metadata back end suffers from significant performance limitations, making it unsuitable for performance-critical installations [59].

Magellan [59] is an on-disk file system that supports scalable searching of metadata. It integrates the search-optimized data structures used by Spyglass within a file system and provides a custom-built search interface, thus providing an end-to-end metadata management framework. However, the approach of integrating metadata management with on-disk file systems lacks modularity.

4.6 Future Work

The new Loris-based metadata management framework opens up several possible avenues for future work. We will outline some possible directions in this section.

4.6.1 Partitioning

Spyglass and Magellan showed how partitioning could be used to achieve scalable search performance in large installations. We are working on implementing a simple *file volume-based* partitioning of the LSM-tree to improve scalability. The fundamental idea behind this partitioning strategy is to maintain a set of LSM-trees, both data and index trees, on a per-volume basis. Since indices are separated on a volume basis, and since users search for files in their own volumes most of the time, query evaluation can be sped up considerably as only the target volume's index is used to find matching files. Other partitioning techniques, like hierarchical partitioning, can also be implemented easily using our infrastructure.

4.6.2 Exploiting heterogeneity

As we explained earlier, the Loris stack can use the attribute infrastructure to exchange policy information between layers. For instance, the naming layer uses this infrastructure to inform the logical layer to mirror directories on all local devices. We plan to use the same infrastructure to assign different files to different types of storage devices. For instance, the Loris files used by the LSM-tree that contains key-value entries representing file system metadata can be stored on an SSD while the secondary indices could be stored on disk drives. Such an approach trades off query performance for space-efficient usage of SSD, as the SSD is used only for serving metadata requests.

4.7 Conclusion

Application-level metadata management subsystems have evolved as a common solution in several application areas to provide scalable metadata indexing and search functionalities lacking in file systems. In this paper, we showed how Loris acts as a modular framework for integrating efficient metadata management subsystems with the storage stack. We presented the design of our Loris-based metadata subsystem and showed how it provides significant performance speedups for metadata intensive workloads. We also showed how our LSM-tree-based indices and attribute-based search interface enable scalable, efficient metadata search.

Chapter 5

Integrating Flash-based SSDs into the Storage Stack

Abstract

Over the past few years, hybrid storage architectures that use high-performance SSDs in concert with high-density HDDs have received significant interest from both industry and academia, due to their capability to improve performance while reducing capital and operating costs. These hybrid architectures differ in their approach to integrating SSDs into the traditional HDD-based storage stack. Of several such possible integrations, two have seen widespread adoption: Caching and Dynamic Storage Tiering.

Although the effectiveness of these architectures under certain workloads is well understood, a systematic side-by-side analysis of these approaches remains difficult due to the range of design alternatives and configuration parameters involved. Such a study is required now more than ever to be able to design effective hybrid storage solutions for deployment in increasingly virtualized modern storage installations that blend several workloads into a single stream.

In this paper, we first present our extensions to the Loris storage stack that transform it into a framework for designing hybrid storage systems. We then illustrate the flexibility of the framework by designing several Caching and DST-based hybrid systems. Following this, we present a systematic side-by-side analysis of these systems under a range of individual workload types and offer insights into the advantages and disadvantages of each architecture. Finally, we discuss the ramifications of our findings on the design of future hybrid storage systems in the light of recent changes in hardware landscape and application workloads.

5.1 Introduction

Over the last decade, flash-based SSDs (Solid State Disks) have revolutionized the storage landscape. Though modern flash SSDs perform much better than their rotating media counterparts under both random and sequential workloads, flash-only storage installations continue to be prohibitively expensive for most, if not all, enterprises due to the high cost/GB of SSDs. As a result, storage researchers have proposed implementing systems based on hybrid storage architectures that use high performance flash SSDs in concert with high-density HDDs (Hard Disk Drives) to reduce capital and operating costs, while improving overall performance.

Of all such architectures, two have gained widespread adoption—*Caching* [2] and *Dynamic Storage Tiering* (DST) [3; 42; 56]. The Caching architecture involves extending the two-level memory hierarchy to the third level by using flash devices as intermediate caches that sit between HDDs and memory. The DST architecture, on the other hand, uses SSDs for primary data storage by establishing tiers of high-performance flash storage and high-density disk storage.

Due to their popularity, these two architectures have also been in the limelight of research over the past few years, and the effectiveness of Caching and DST under certain specific workloads is well understood [37; 53]. However, with the wide spread adoption of storage virtualization, modern storage installations blend I/O requests from different workloads together into a single stream. Designing efficient hybrid architectures for such workloads requires answering two important questions: 1) how do existing architectures fare under such workloads?, and 2) should future hybrid systems support not one, but multiple architectures, and pair workloads with their ideal architectures?

In order to answer these questions, we need to perform 1) a side-by-side comparison of existing architectures under such mixed workloads, and 2) a systematic study of interactions between architectural design alternatives and workload parameters. Unfortunately, due to the wide range of configuration parameters and design alternatives involved in building Caching and DST-based systems, performing such a study would be infeasible in the absence of a hybrid storage framework.

In this paper, we will show how the Loris storage stack, with a few extensions, can be transformed into a modular framework for implementing and evaluating hybrid storage systems. To illustrate the flexibility of the framework, we will implement several flavors of Caching and DST. We will then use several macrobenchmarks and file system workload generators to perform a systematic study of the effectiveness of these Loris-based hybrid systems under a variety of workloads. Based on our evaluation, we will offer insights into 1) the design of current hybrid systems by investigating design factors that impact performance, and 2) the design of future systems in light of recent changes in hardware landscape and

application workloads.

The rest of the paper is organized as follows. In Sec. 5.2, we will present a classification of Caching and DST architectures based on several design parameters. In Sec. 5.3, we will introduce the Loris stack and describe the plugin-based extensions that transform it into a hybrid storage framework. Following this, we will describe how we used this framework to implement Loris-based Caching and DST systems in Sec. 5.4. We will then present our side-by-side evaluation of these hybrid systems using several benchmarks in Sec. 5.5. Finally, we will discuss the ramifications of our findings in Sec. 5.6, and conclude in Sec. 5.7.

5.2 Hybrid storage systems

As we mentioned earlier, Caching and DST architectures differ in the way they integrate SSDs into the HDD-based traditional storage stack. In this section, we will explore the design space of these hybrid architectures and classify them based on several design parameters.

5.2.1 Caching

Caching architectures use SSDs as a non-volatile, intermediate caches between the system memory (RAM) and HDDs. Thus, in all Caching architectures, SSDs contain only cached copies of HDD-resident primary data.

Based on when data is cached, Caching architectures can be classified as *On-demand* or *Interval-driven*. While data is cached as a side effect of a read operation is On-demand Caching, Interval-driven Caching monitors data blocks and periodically, once every preconfigured interval, trades old SSD-resident "cold" data for new HDD-resident "hot" data.

Irrespective of when data is cached, Caching architectures can be classified as *read-only* or *read-write* caches depending on their behavior with respect to write operations. Read-only caches maintain only clean data. Thus, writes to uncached data blocks are not buffered by the SSDs, and updates to cached data blocks invalidate the cached copies. ZFS's L2ARC [62] and NetApp's FlashCache [2] are examples of read-only caches used to speed up workloads dominated by random reads.

Read-write caches, on the other hand, cache both data reads and writes. They can be further classified into *Write-back* and *Write-through* caches. A Write-back cache eliminates all foreground HDD writes by buffering them in the SSD and resynchronizing the primary HDD copy later. Since the cached SSD copy and primary HDD copy can be out of sync in a Write-back Caching system, extra bookkeeping is required to maintain consistency and prevent data loss across power failures or system reboots. EMC's FastCache [5] is an example of a Write-back

cache that uses flash drives configured as RAID1 mirror pairs to guarantee reliability in the face of system or power failures.

A Write-through cache, on the other hand, forwards writes to both the cached SSD copy and the primary disk copy. By maintaining all data copies in sync, Write-through caching avoids additional (potentially synchronous) metadata updates at the expense of foreground write performance. One could further classify a Write-through cache into a *Write-through-all* cache or *Write-through-update* cache depending on how writes to uncached blocks are handled. While a Write-through-all cache admits uncached data blocks, a Write-through-update cache sieves new data by admitting only cached data writes. Azor [53] is an example of a Write-through Caching system that supports both Write-through-all and Write-through-update Caching.

5.2.2 Dynamic Storage Tiering

Dynamic Storage Tiering architectures (DST) organize high-performance, flash-based SSDs and high-density, magnetic HDDs into multitier systems and partition data between tiers depending on several price, performance, or reliability factors. Thus, unlike Caching architectures, each data item in a DST system is stored in only one location.

Based on the initial allocation policy used, DST architectures can be classified into *Hot-DST* and *Cold-DST* types. With Hot-DST architectures, data is initially allocated on the HDD tier. Periodically, "hot" data are migrated to the SSD tier. With "Cold-DST" architectures, data are initially allocated on the SSD tier and "cold" data are periodically demoted to the HDD tier. IBM's EasyTier [96], Compellent's tiering systems [4], EDT [37] and HyStor [19] are a few examples of Hot-DST systems. Hot-DST architectures can be further classified depending on the time at which "hot" data are migrated. In *Dynamic Hot-DST* systems, "hot" data from the HDD tier are migrated on demand, while in *Interval-driven Hot-DST* systems, data are migrated at predefined intervals. Almost all Hot-DST systems we are aware of are interval driven.

At a very high level, Hot-DST and Caching architectures appear to be identical with respect to their mode of operation. Both Interval-driven architectures migrate/cache data at periodic intervals. Both On-demand architectures migrate/cache data as a side effect of a read operation. Furthermore, in order to be able to map data to their ideal storage targets, both Caching and DST architectures observe access patterns and classify data as "hot" or "cold." For instance, all Caching architectures use a second-level caching algorithm (like L2ARC [62]) and all DST architectures use some "hot" data identification mechanism (like inverse bitmaps [19]), to identify "hot" data that must be serviced by the SSDs. This raises two questions: 1) can popular DST algorithms be used for implementing efficient Caching architectures and vice versa?, and 2) all other factors considered

identical, is there a performance impact associated with the most important design difference—presence or absence of a data copy?

Later, in Sec. 5.4, we will address the first question by showing how we use a popular DST algorithm to implement efficient Caching systems. Then, in Sec. 5.5, we will evaluate the Caching and DST implementations side by side to answer the second question. Having described several hybrid architectures, we will now give a brief overview of the Loris stack and show how we use it as framework to implement hybrid systems.

5.3 Background: The Loris storage stack

In prior work, we proposed Loris [9], a redesign of the storage stack. Loris is made up of four layers as shown in Figure 5.1. The interface between these layers is a standardized file interface consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. Every Loris file is uniquely identified using a <volume identifier, file identifier> pair. Each Loris file is also associated with several *attributes*, and the interface supports two attribute manipulation operations—*getattribute* and *setattribute*. Attributes enable information sharing between layers, and are also used to store out-of-band file metadata. We will now briefly outline the responsibilities of each layer in a bottom-up fashion.

5.3.1 Physical layer

The physical layer is tasked with providing 1) device-specific layout schemes, and persistent storage of files and their attributes, 2) end-to-end data verification using parental checksumming, and 3) fine-grained data sharing and individual file snapshoting. Thus, the physical layer exports a snapshotable physical file abstraction to the logical layer. Each storage device is managed by a separate instance of the physical layer, and we call each instance a *physical module*.

5.3.2 Logical layer

The logical layer provides both device and file management functionalities. It is made up of two sublayers, namely the file pool sublayer at the bottom, and the volume management sublayer at the top. The logical layer exports a *logical file* abstraction to the cache layer. A logical file is a virtualized file that appears to be a single, flat file to the cache layer. Details such as the physical files that constitute a logical file, the RAID levels used, etc. are confined within the two sublayers. We will now briefly describe the functionalities of each sublayer.

The volume management sublayer is responsible for providing both file volume virtualization and per-file RAID services. It maintains data structures that track the membership of files in file volumes and mapping between physical files

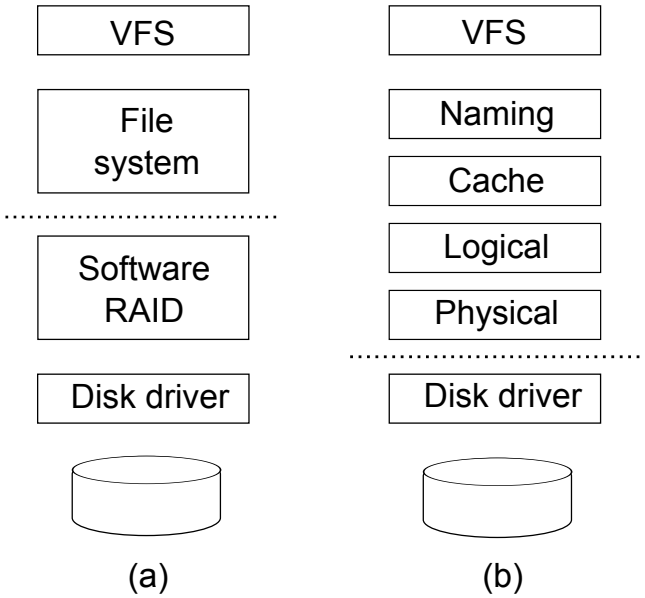


Figure 5.1: This figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

and logical files. It also provides file management operations that enable snapshoting and cloning of files or file volumes. In prior work, Loris has been used to design a new storage model [10]. File pools simplify storage administration and enable thin provisioning of file volumes [10]. The file pool sublayer maintains data structures necessary for tracking device memberships in file pools, and provides device management operations for online addition, removal and hot swapping of devices.

Each file volume is represented by a *volume index file* that tracks logical files belonging to that volume. The volume index is created during volume creation, and it stores an array of entries containing the *configuration information* for each logical file in that volume. This configuration information is 1) the RAID level used, 2) the stripe size used, and 3) the set of physical files that make up the logical file. Similar to the way files are tracked by the volume index file, file volumes themselves are tracked using the *meta index file*. This file also contains an array of entries, one per file volume, containing *file volume metadata*. Thus, using these two data structures, the volume management sublayer supports file volume virtualization. Multiple file volumes can be created in a single file pool in Loris which makes thin provisioning of file volumes possible.

5.3.3 Cache and Naming layers

The cache layer provides data caching. As the cache layer is file-aware, it can provide different data staging and eviction policies for different files or types of files.

The naming layer acts as the interface layer. Our prototype naming layer implements the traditional POSIX interface. The naming layer uses Loris files to store data blocks of directories that contain directory entries. It also uses the attribute infrastructure in Loris to store POSIX attributes of each file as Loris attributes. All POSIX semantics are confined to the naming layer. For instance, as far as the logical layer is concerned, directories are just regular files.

5.3.4 Tiering Framework

All hybrid storage systems, irrespective of how they integrate flash into the storage stack, essentially attempt to pair data with their ideal storage device to maximize performance. In order to do so, all these systems have to 1) collect and maintain access statistics to classify data, and 2) implement background migration to transparently relocate data to their designated target. We had to extend Loris to support these two functionalities in order to transform it into a framework.

We did this by extending Loris' logical layer. There were three main reasons for extending the logical layer compared to other layers. First, the logical layer uses the logical file abstraction to implement per-file RAID algorithms by multiplexing requests across physical files. We can exploit the same abstraction to support transparent migration of files between devices/physical modules. Second, the logical layer has information about both file access patterns and device performance characteristics, making it the ideal spot for implementing algorithms that take into account both these factors. Third, access statistics collected at the logical layer reflect the real storage workload after caching effects have been filtered out. Thus, by confining changes to the logical layer, we modularly extend the Loris stack to implement hybrid systems without affecting algorithms in any of the other layers.

Data collection plugin

Several DST and Caching systems have proposed collecting different access statistics for classifying data. For instance, EDT [37] is a DST system designed for installations that consist of SSD, SAS (Serial Attached SCSI) and HDD tiers. For achieving optimal performance under such multitier installation, EDT classifies hot extents into IOPS-heavy and bandwidth-heavy types, and stores IOPS-heavy extents on the SSD tier and bandwidth-heavy extents on the SAS tier. Thus for implementing an EDT-style DST system, one must collect and maintain IOPS and

bandwidth requirements for each file. Azor [53], on the other hand, is an SSD-based Caching system that maintains access frequencies for each cached SSD block and uses it to perform cache admission control. Thus for implementing an Azor-style Caching system, one must maintain access counts for each file.

To support multiple such design alternatives, we extended the logical layer using a generic plugin model. The *data collection plugin* is responsible for collecting and maintaining access statistics for each file. Each plugin implementation is required to support a standard set of callback routines. During the startup phase, depending on the type of hybrid configuration to be deployed, the appropriate data collector is registered with the logical layer, which then invokes the callback routines at strategic points during execution.

We implemented a data collection plugin that uses inverse bitmaps [19] to identify performance-critical files that should be cached or migrated. During every read and write operation, the inverse bitmap b is calculated as shown below.

$$b = 2^{6-\lfloor \log_2(N) \rfloor} \tag{5.1}$$

In the equation, N refers to the number of 4-KB pages read/written from the file. The value 6 is an implementation-specific constant chosen based on the maximum number of 4-KB file pages read or written by the cache layer in a single operation (64). The computed value is then added to a 32-bit counter associated with that file. Thus, the inverse bitmap assigns a large weight to files read/written in small chunks, which could either be small files or large files randomly read/written in small chunks, thereby prioritizing random accesses over sequential ones.

Our current prototype maintains an in-memory array of counters, one per tier. When queried for the "hottest" file in the disk tier, the plugin picks the most recently used file with the highest counter value. When queried for the "coldest" file in the SSD tier, the plugin picks the least recently used file with the lowest counter value. Thus, we extend the original inverse bitmap design [19] by using recency as a tiebreaker among files with identical counter values. The in-memory approach is probably not scalable as modern installations consist of millions of files, so we are currently considering using priority dequeues using external heap variants or dynamic histograms with delayed updates for scalable maintenance of access statistics.

Our data collection plugin also explicitly keeps track of the counter value of the last file that has been evicted from the SSD tier during cleanup. It uses this value to perform admission control. A file is qualified for migration to the SSD tier only if its counter value is higher than that of the last evicted file.

We would like to point out here that all hybrid architectures we present in Sec. 5.4 use the inverse bitmap plugin as their data collector. Thus, although inverse bitmaps were originally introduced and used in Hystor [19] for DST, we show how it can also used to implement high performance Caching architectures.

Thus, as we mentioned earlier, most data collection algorithms are architecture neutral.

We would also like to point out that although we do not consider multitier installations (such as ones including SAS drives) as a part of this work, extending Loris to such configurations only requires replacing relevant plugins.

File migration

As we mentioned earlier, we exploited the logical file abstraction of the logical layer to support transparent file migration between physical modules. Our current implementation locks each file during migration to prevent foreground requests from accessing the source during migration. We are also working on implementing transparent, incremental migration of file data. The incremental migration implementation would first take a snapshot of the target file using the individual file snapshoting functionality present in Loris, following which it would copy the snapshot's data and attributes to the designated target. After successfully copying the snapshot, the migration plugin would then, if need be, perform an incremental transfer of data modified since the snapshot.

5.3.5 Loris as a platform for storage tiering - The Pros

There are several advantages in using Loris as the basis for implementing DST solutions. First, most DST solutions exploit device heterogeneity to improve performance. For instance, Avere's DST system [3] stores all write-only files on the SAS tier using a log-structured layout to optimize write throughput. Since the Loris stack provides the capability to pair devices with their ideal layout algorithms, it can be used to exploit heterogeneity inherent in tiered systems.

Second, several DST systems use semantic information to identify crucial data (like file system metadata). As we mentioned earlier, semantic information is exchanged between layers in the Loris stack using the attribute infrastructure. In the Loris stack, each file create carries with it a file type attribute that informs the logical layer if the file is a metadata file (directory) or a data file. Thus, the logical layer can use this semantic information to assign different policies to files or file types. We will show later how we use semantic information to implement 1) *type-aware sieving* of large files, and 2) per-file tiering policy later in the paper.

Third, administrative operations like hot-swapping and online addition and removal of devices are mandatory features in any enterprise DST system. The file pool model in Loris simplifies storage administration and is capable of supporting all these features.

5.3.6 Loris as a platform for storage tiering - The Cons

Since Loris' logical layer maintains mapping information at the granularity of whole files, implementing Caching or DST systems that operate on a sub-file basis is not possible. Consider an append write to an uncached file for instance. In order to implement Write-back Caching, Loris would have to buffer this write in the SSD. Doing so would require the logical layer to map two sets of logical file offsets to two different physical files (offset range <0, old file size $-1>$ to physical file stored in HDD, and range $<$old file size, new file size $-1>$ to a physical file on the SSD). The current mapping infrastructure only supports mapping a whole logical file to one or more physical files.

However, we would like to emphasize the fact that this is a limitation of just the current implementation. We are working on designing a new mapping format for the logical layer that supports sub-file mapping. With the new infrastructure, Loris would select the mapping type on a per-file basis. For instance, while all small files and files read/written in their entirety could be stored in a mapping file based on the old format, a file that is read/written in 4-KB chunk could use the new format that could potentially map each 4-KB logical block to a different physical file. By using this extension mapping infrastructure, we intend to evaluate block, extent and file-level tiering and Caching implementations side by side as a part of future work.

We would also like to point out that despite the lack of subfile mapping capability in Loris, all the results we present in this work are equally applicable to block or extent-level implementations. As all Caching and DST architectures are implemented using a single framework, and as all of them share the same data collection plugin (which maintains access statistics at the granularity of whole files), we believe that a block or extent-based realization of these architectures, under similar workloads, using the same access statistics would produce comparative results identical to our study.

5.4 Loris-based hybrid systems

Having described the plugins, we will now show how we use these plugins to implement several Caching and DST systems for a two-tier (SSD/disk) installation. Common to all these systems is the *type-aware sieving* of large files. During preliminary evaluation of these hybrid systems, we found out that certain benchmarks (Web Server) create large, append-only log files that were never read. As these files received a lot of writes, their bitmap counter values were high. As a result all hybrid systems pinned these log files to the SSD tier, thereby wasting valuable space that could be used for housing other "genuinely hot" files. To prevent this, we added type-aware sieving to Loris. With sieving, any file larger than a configurable threshold, which in our current prototype is 1-MB, will not be cached or

migrated to the SSD. Similar, any SSD-resident file is explicitly demoted (in the case of DST) or invalidated (in the case of Caching) when it grows beyond 1-MB. Type-aware sieving is an example of how we use semantic awareness of the Loris stack to improve the performance of all hybrid systems.

Also common to all these systems is the cleaner implementation. A cleanup of the SSD tier is triggered when a write operation to the SSD tier cannot be completed due to lack of space. This can happen either during a foreground write operation to a file in the SSD tier, or during background migration/caching of a file from the disk tier. In both cases, the cleaner consults the data collection plugin to determine the set of cold files to evict from the SSD tier. The action taken by the cleaner depends on the architecture being implemented. For Caching architectures, the cleaner simply invalidates the cached file copy by deleting it from the SSD. For DST implementations, the cleaner invokes the migration plugin, demoting those files back to the disk tier. This cold migration continues until enough space has been cleared to finish the write operation. In addition to such foreground cleaning, we also run the cleaner in the context of a background thread, to proactively clean the SSD tier, under certain hybrid configurations as we will show later.

We will now describe how we implemented several hybrid storage systems using the Loris stack.

5.4.1 Loris-based Hot-DST systems

We will now describe the Loris-based implementation of two Hot-DST architectures. As explained in Sec. 5.2, all Hot-DST architectures allocate data on the HDD tier. They differ based on when they migrate "hot" data to the SSD tier.

Dynamic Hot-DST

Our Dynamic Hot-DST implementation migrates "hot" files as a side effect of a read operation that is serviced by the HDD tier. It first queries the data collection plugin to determine if the file is a valid migration candidate. As we mentioned earlier, our data collection plugin considers a file to be a valid candidate if its counter value is higher than that of the file last evicted from the SSD tier. In such a case, the DST implementation queues the file for migration with the migration plugin.

Interval-driven Hot-DST

We also implemented a system based on the Interval-driven Hot-DST architecture. Every preconfigured number of seconds, our Hot-DST implementation runs in the context of a background thread and migrates "hot" files identified by the data collection plugin to the SSD tier. Hot file migration continues until all potential

candidates have been migrated or the "hottest" file in the disk is colder (has a lower counter value) than the "coldest" file in the SSD tier. The data collection plugin verifies the latter condition by comparing the next candidate file's access counter with that of the file last evicted from the SSD.

As the interval of migration is a configuration parameter, we will use two versions (five and eighty seconds) of our Interval-driven Hot-DST system to evaluate the impact of migration interval on overall performance.

5.4.2 Loris-based Cold-DST architectures

We will now describe the Loris-based implementation of three Cold-DST architectures. As explained in Sec. 5.2, all Cold-DST architectures allocate data on the SSD tier.

Plain Cold-DST

This is conceptually the simplest of all DST implementations. This system does not perform any form of "hot" file migration. Foreground write requests to files in the SSD tier that are unable to complete due to lack of space automatically trigger tier cleanup. Files that are demoted during cleaning are never migrated back to the SSD tier, not even if they become "hotter" at a later point in time. Thus, this implementation uses the data collection plugin only to determine which files to evict from the SSD tier.

Dynamic Cold-DST

While the Plain Cold-DST system would work well with workloads where newly created data accounts for a significant fraction of accesses, it would perform poorly under workloads with shifting locality. This is because any access to data that has been "cold" migrated will be serviced by the disk tier.

We solve this problem by adding on-demand "hot"-file migration to the Plain Cold-DST system. Similar to the Dynamic Hot-DST system, data-collector-approved files are migrated in the background as a side effect of a read operation that finds the file in the HDD tier. However, unlike the Hot-DST counterpart, new files continue to be allocated on the SSD tier in the Dynamic Cold-DST system.

Dynamic Cold-DST with background cleaning

When operating with a full SSD, dynamic migration and foreground write requests trigger tier cleanup. As cleaning requires migrating "cold" files off the SSD tier, these writes blocks until sufficient free space has been generated. To avoid stalling write requests, we added a proactive background cleaner to our Dynamic Cold-DST implementation. The cleaner implementation maintains a running counter

of the total amount of "cold" data evicted from a full SSD tier as a side effect of foreground or dynamic migration writes. It uses this counter as an estimate of the amount of space to recover for speeding up future write operations. The cleaner runs in the context of a background thread and starts evicting "cold" files as a side effect of the first blocking write request.

5.4.3 Loris-based Caching

As we explained earlier, the whole-file mapping infrastructure of our current prototype makes it impossible to implement Write-back or Write-through-all Caching systems. We will now describe the implementation of two Write-through-update Caching architectures.

On-demand Caching

On-demand Caching, for most part, works similar to Dynamic Hot-DST system. with the only difference being the fact that files are cached rather than being migrated. As the primary file copy continues to reside in the HDD, unlike the Hot-DST counterpart, cleaning the SSD only requires deleting "cold" files to invalidate them. As cleaning can happen as a side-effect of foreground write operation, and as On-demand Caching is identical to Dynamic Hot-DST in every other aspect, we can compare the two implementations head-to-head to measure the SSD cleaning overhead.

Interval-driven Caching

Interval-driven Caching works identical to the Interval-driven Hot-DST implementation with the exception that files are copied rather than migrated. Similar to its Hot-DST counterpart, all files are initially allocated on the HDD. Periodically, at a configurable interval (five seconds in our current prototype), the statistics accumulated by the data collector are used to cache "hot" files in the SSD tier. Similar to the Dynamic Hot-DST—Caching comparison, we can also compare the two Interval-driven architectures to measure the impact of cleaning on overall performance.

5.5 Evaluation

Having described how we implemented various hybrid architectures, we will now present a systematic analysis of the effectiveness of these architectures under a wide range of workloads. We will first describe the hardware setup and benchmarking tools we used for our evaluations. We will then present a side-by-side

evaluation of these architectures and offer insights into the interaction between design alternatives and workload parameters.

5.5.1 Test Setup

All tests were conducted on an Intel Core 2 Duo E8600 PC, with 4-GB RAM, using a 500-GB 7200-RPM Western Digital Caviar Blue SATA hard disk (WD5000AAKS), and a OCZ Vertex3 Max IOPS SSD. Table 5.1 lists the performance characteristics of these devices. We ran all tests on 8-GB test partitions at the beginning of the devices.

Device	BB/s	Random 4K IOPS
WD5000AAKS	126/126	112/91
OCZ Vertex 3 MAX IOPS	550/500	35000/75000

Table 5.1: Device properties: The table lists the read/write performance characteristics of the two SATA devices we use for evaluating various hybrid architectures.

The Loris prototype has been implemented on the MINIX 3 multiserver operating system [44]. We deliberately configured Loris to run with a 64-MB data cache to ensure that the working set generated by benchmarks is at least an order of magnitude larger than the in-memory cache. Thus, using a low cache size, we heavily stress the I/O subsystem.

To estimate the effect that SSD size has on the effectiveness of various DST models, we modified the Loris stack to keep track of the amount of user data written to the SSD and used it to artificially limit the available SSD size. We used this to evaluate the effectiveness of various architectures at three different SSD sizes, namely, 25%, 50%, and 75% of the total working set size. We determined the working set size by running each workload using a disk-only configuration.

We are aware of the fact that certain consumer grade SSDs exhibit performance deterioration when occupied at 100% capacity. Although our artificial approximation of available SSD size sidesteps this issue, we believe that the results we derive are still applicable as we target enterprise installations that are likely to use enterprise-grade SSDs. Unlike consumer-grade SSDs, these high-end SSDs are known to overprovision large amounts of scratch space to avoid the write-cliff phenomenon [41].

5.5.2 Benchmarks and Workload Generators

Since we wanted to systematically analyze the interactions between architectural design alternatives and workload parameters, we used Postmark and FileBench to generate four different classes of server workloads. Postmark is a widely used,

configurable file system benchmark that simulates a Mail Server workload. It performs a specified number of transactions, where each transaction pairs a whole-file read or append operation with a create or delete operation. We report the transaction time, which excludes the initial file preallocation phase, for all hybrid systems.

FileBench is an application-level workload simulator that can be used to model workloads using a flexible workload modeling language (WML). We used two predefined workload models to generate File Server and Web Server workloads. We ran each workload for half an hour and we present the IOPS reported by FileBench for all hybrid systems. Preliminary evaluation revealed wide variations in results across different FileBench runs (even with the same hybrid architecture). We traced this back to the variation in random seed selection between runs. In order to reliably compare different hybrid architectures, we modified FileBench to use a fixed random seed across all runs. With this patch, all the results we obtained were reproducible.

5.5.3 Workload Categories

Using Postmark and FileBench, we were able to vary a variety of workload parameters like file size distribution, read-write ratios, access patterns (sequential vs. random) and access locality (random vs Zipf). We will now detail the properties and configuration parameters of each workload.

Mail Server

We configured Postmark to perform 80,000 transactions on 40,000 files, spread over 10 subdirectories, with file sizes ranging from 4-KB to 28-KB, and read/write granularities of 4-KB. The resulting workload is dominated by small file accesses, has a 1:2 read-write ratio, and exhibits random access pattern with very little locality.

File Server

We configured FileBench to generate 10,000 files, using a mean directory width of 20 files. The median file size used is 128-KB, which results in files an order of magnitude larger than the rest of the workloads. The workload generator performs a sequence of create, write, read, append (using a fixed I/O size of 1-MB), delete and stat operations, resulting in a write-biased workload. As all files are read in their entirety, and as append operations are large, the access pattern is sequential. The workload lacks locality as file access distribution is uniform.

Web Server

The Web Server configuration generates 25,000 files, using a mean directory width of 20 files. The median file size used is 32-KB, which results in a workload dominated by small file accesses, with the only exception being an append-only log file. The workload generator performs a sequence of ten whole-file read operations, simulating reading web pages, followed by an append operation (with an I/O size of 16-KB) to a single log file. This results in a 10:1 read-write ratio unlike other benchmarks. Though files are read in their entirety, the small file size results in the file access pattern being essentially random. Similar to the File Server workload, this workload also lacks locality due to the uniform file access distribution.

Web Server (Zipf Distribution)

All the workloads described above lack locality which is present in several real-life workloads. For instance, while the Web Server workload generated by FileBench lacks locality, it is well known that file accesses in web servers tend to follow a Zipf distribution [38]. Thus, we modified FileBench to generate a Zipf-based file access distribution. In addition, we also wanted to evaluate the effectiveness of Dynamic Cold-DST architecture under workloads with shifting locality. To do so, we modified the default Web Server workload to generate two sets of 25,000 files instead of one. As a result, the first fileset get flushed out to the HDD tier as a part of cleaning up the SSD tier to accommodate the second one. We then ran the workload generator on the first set, thereby simulating shifting workload locality - the adversarial case for the Plain Cold-DST architecture.

5.5.4 Comparative evaluation

Having described the workload parameters, we will now present our evaluation of various hybrid systems. We will first analyze Loris-based DST systems to identify the impact of architecture-specific design alternatives (Sec. 5.5.4, Sec. 5.5.4). We will then compare DST systems with their Caching counterparts to identify top performers under various workloads (Sec. 5.5.4). Finally, we will present the side-by-side comparison of these top performers under a mixed workload that simulates a virtualized workload (Sec. 5.5.5).

Hot-DST Architectures

Figures 5.2, 5.3, 5.4, 5.5 show the performance of various Hot-DST configurations. There are a number of interesting observations to be made from the results.

First, as can be seen in Figure 5.5, locality plays a major role in deciding the effectiveness of Hot-DST architectures. Even at SSD sizes covering as little as

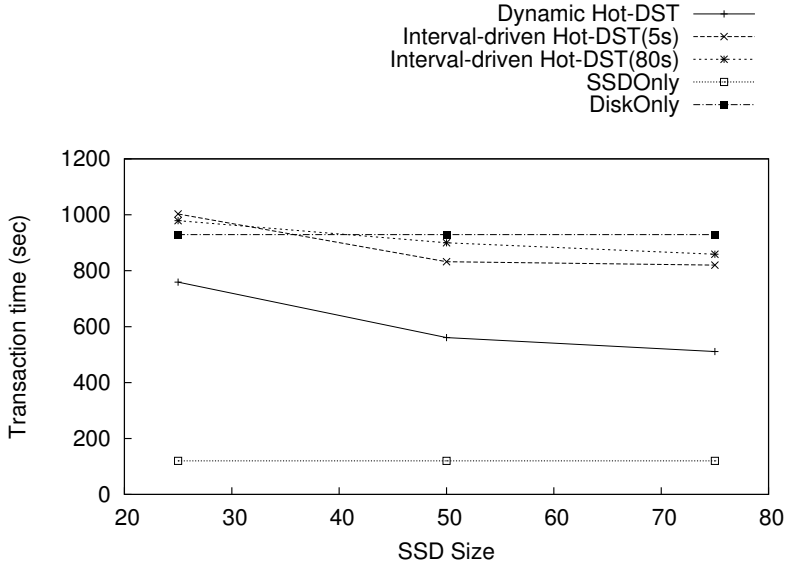


Figure 5.2: Transaction time (seconds) under Postmark for various Hot-DST architectures.

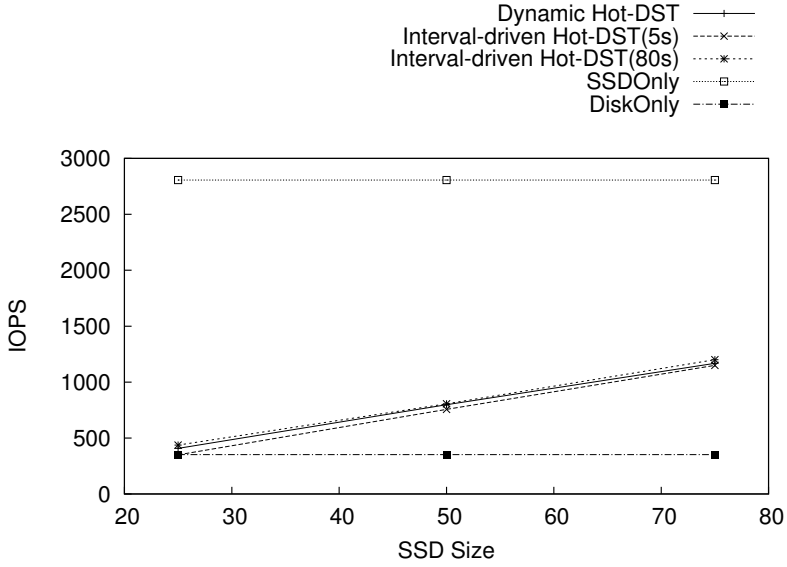


Figure 5.3: IOPS delivered under FileBench's File Server workload by various Hot-DST architectures.

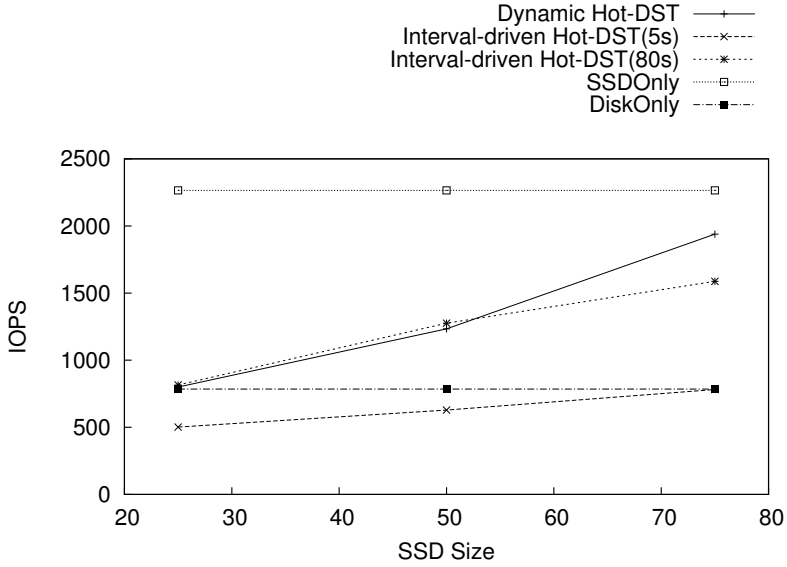


Figure 5.4: IOPS delivered under FileBench's Web Server workload by various Hot-DST architectures.

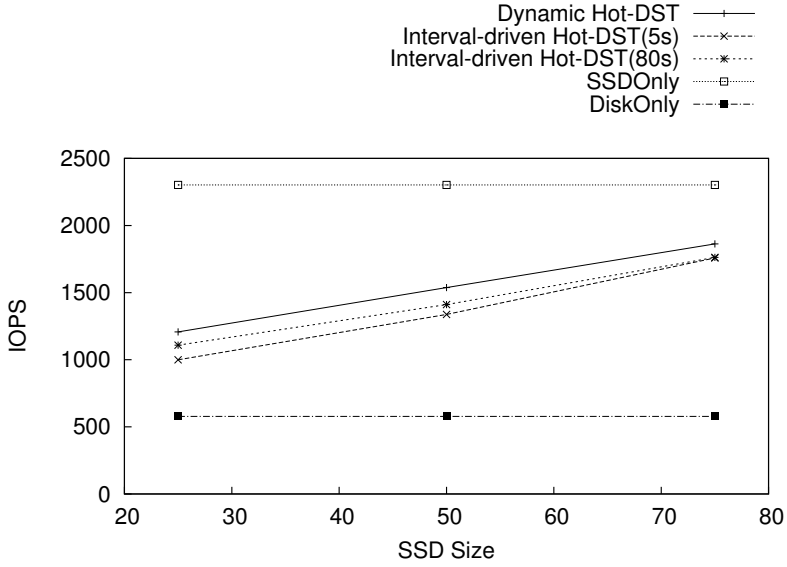


Figure 5.5: IOPS delivered under FileBench's Web Server (with Zipf) workload by various Hot-DST architectures.

25% of the total working set, all Hot-DST configurations show significant performance improvement compared to the HDD-only case.

Second, under all workloads, the migration interval has a significant impact on overall performance. For instance, under the Web Server workload (Figure 5.4), the five-second, Interval-driven Hot-DST system delivers lower IOPS than even the HDD-only case. The eighty-second Hot-DST system, on the other hand, improves performance significantly. As we mentioned earlier, the Web Server and File Server workloads access files uniformly in sequence without any locality. As a result, when we traced migration patterns at the logical layer, we found quite a large number of files shuttling back and forth between tiers, with the number of such files increasing as the migration interval decreases. Thus, the thrashing of files caused by the workload's uniform access pattern destroys performance by interfering with foreground reads and writes serviced by the HDD tier.

Third, unlike other workloads, the eighty-second, Interval-driven Hot-DST performs poorly under Postmark. At low SSD sizes, it takes longer than the HDD-only configuration to finish the transactions. Even at high SSD sizes covering as much 75% of the working set, it performs only slightly (12%) faster. Analysis revealed that this was due to two factors. First, as we mentioned earlier, the workload generated by Postmark lacks locality. Thus, it is adversarial in nature for locality-dependent algorithms like Interval-driven Hot-DST. Second, unlike FileBench, Postmark is transaction-bound rather than time-bound (the termination condition is a limit on the number of transactions). Although we perform 80,000 transactions, analysis revealed that the total transaction time was not long enough for Interval-driven Hot-DST to stabilize.

Fourth, while Dynamic Hot-DST performs similar to other Hot-DST architectures under File Server (Fig. 5.3) and Web Server (Fig. 5.4) workloads, it provides significant improvement under Postmark (Fig. 5.2). This is unexpected, especially given the fact that the aforementioned reasons that slow down Interval-driven Hot-DST also apply to Dynamic Hot-DST. By profiling the system, we found the source of this performance improvement to be a counterintuitive increase in the number of writes serviced by the SSD tier under Dynamic Hot-DST. As we mentioned earlier, Postmark pairs creates/deletes with reads/writes. Writes issued by Postmark append data to existing files. As these append operations are typically not block aligned, they trigger an "append-read" operation to fetch the append target (the file's last data block). The ensuing "append write" is then buffered by the data cache (caching layer). As a side effect of this read operation, the target file's access counter gets updated by a large increment (due to the small amount of data being read), which results in the file being migrated to the SSD. When the data is flushed from the cache at a later time, it gets serviced by the SSD (which now hosts the "hot" file) resulting in the performance improvement. Interval-driven DST architectures do not benefit from these "append-reads" as they miss the window of opportunity due to not performing on-demand migration. We verified that this

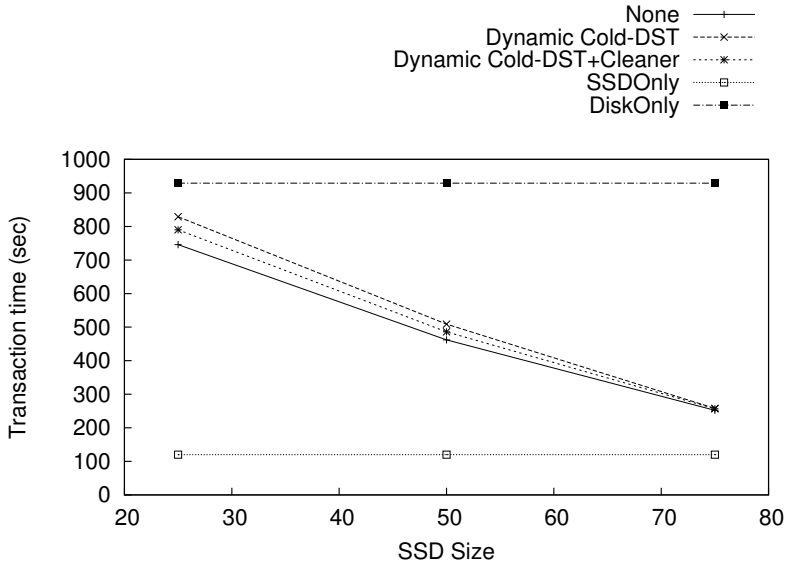


Figure 5.6: Transaction time (seconds) under Postmark for various Cold-DST architectures.

was indeed the case by not updating access statistics on append reads. Under such circumstances, the Interval-driven Hot-DST outperformed Dynamic Hot-DST.

Cold-DST Architectures

Figures 5.6, 5.7, 5.8, 5.9 show the performance of various Cold-DST configurations. As can be seen in Figure 5.9, the adversarial workload with locality results in Plain Cold-DST performing poorly when compared to the Dynamic Cold-DST architecture. An interesting observation is that the Plain Cold-DST architecture still performs noticeably better than the HDD-only case. On investigating this, we found that during the preallocation phase, directories receive quite a lot of read/write accesses. These accessess cause directories to possess high counter values in comparison to other files. As a result, despite the barrage of creates and writes during the preallocation phase, directories remain pinned to the SSD tier. Thus, all directory accesses during the workload run are serviced by the SSD causing a noticeable performance improvement over the HDD-only case.

If we consider workloads without locality, we see that the Dynamic Cold-DST architecture deteriorates performance compared to Plain Cold-DST. Due to the lack of locality, the performance gained by servicing reads from the SSD tier does not match the overhead of migrating "hot" data from the HDD tier. However, an interesting observation is how, despite similar access patterns (uniform without

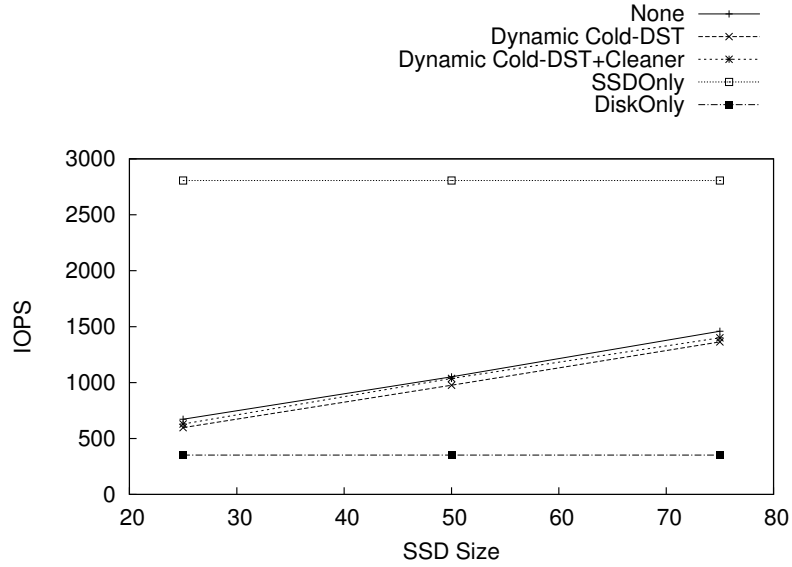


Figure 5.7: IOPS delivered under FileBench's File Server workload by various Cold-DST architectures.

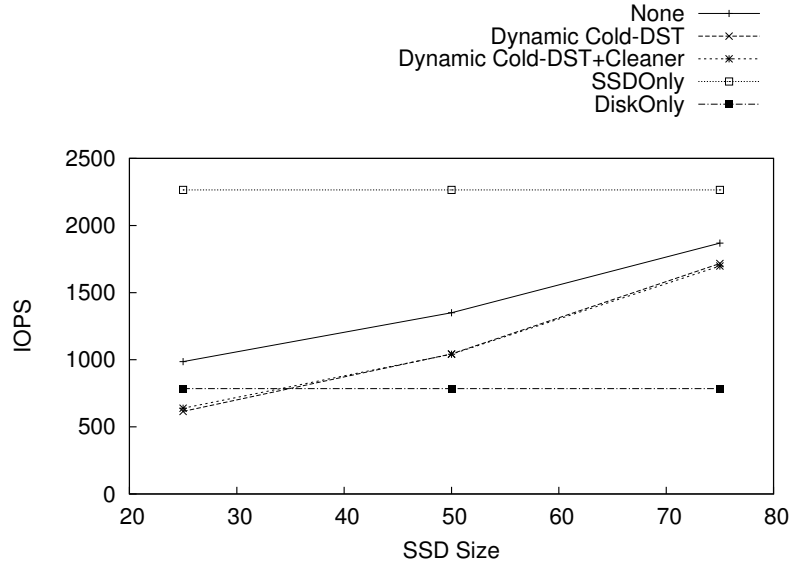


Figure 5.8: IOPS delivered under FileBench's Web Server workload by various Cold-DST architectures.

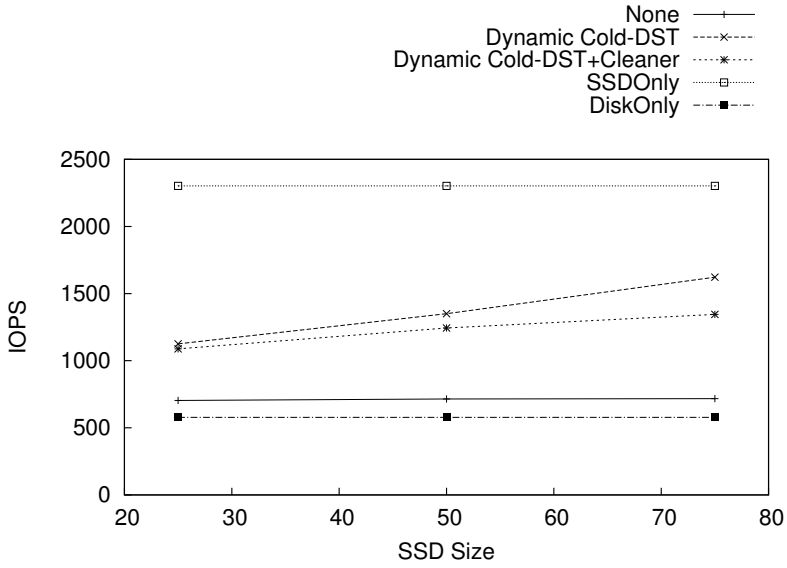


Figure 5.9: IOPS delivered under FileBench's Web Server (with Zipf) workload by various Cold-DST architectures.

locality), File Server and Web Server workloads produce different comparative results. While Dynamic Cold-DST catches up with Plain Cold-DST under the File Server workload (Figure 5.7), it continues to lag behind by a huge margin under the Web Server workload (Figure 5.8).

On investigating this further, we found that the delete operations performed by the File Server workload play an indirect, albeit crucial, role in improving the overall performance of Cold-DST architectures. Logically speaking, creating new files in an already full SSD tier should have an adverse impact on performance as these writes cannot be completed without evicting "cold" data. However, in reality, newly written data is first buffered by Loris' data cache (cache layer) before being flushed out to the SSD. Unlike write operations, file deletes propagate down through the layers immediately. These delete operations free up space in the SSD tier indirectly accelerating both delayed foreground writes and background dynamic migrations. As the performance gained by allocating new files on the SSD tier offsets the performance drop caused by migrating "hot" files, Dynamic Cold-DST catches up with Plain Cold-DST under the File Server workload.

We saw only marginal improvement (at best) from adding background cleaning to the Dynamic Cold-DST architecture. We believe that this is due to our cleaner being overly conservative in freeing up space. During experimentation, we found out that aggressive cleaning had a negative impact on performance under most workloads we used in this study. However, recent analysis of network

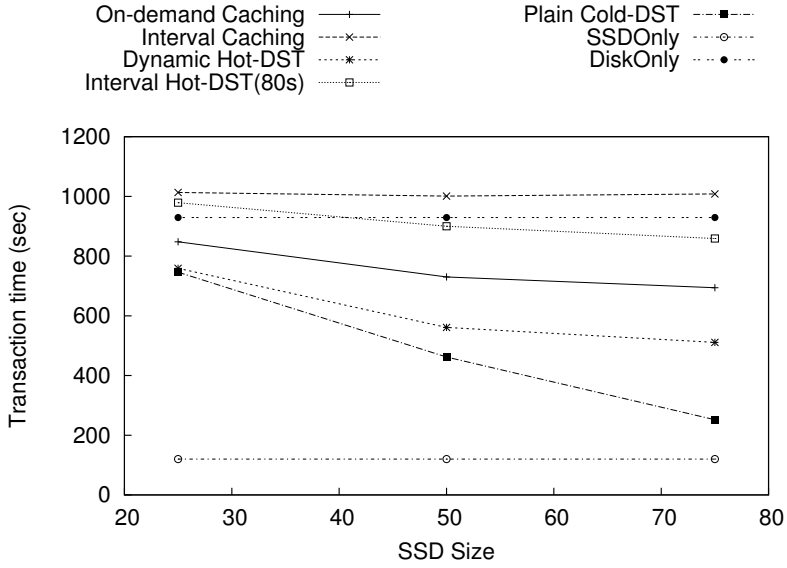


Figure 5.10: Transaction time (seconds) under Postmark for Caching and DST architectures.

file system traces indicate that over 90% of newly created files are opened less than five times [61]. Under such conditions, a Dynamic Cold-DST implementation would definitely benefit from aggressive cleaning. We intend to perform a reevaluation of our Cold-DST implementations under trace-driven workloads as a part of future work.

DST vs Caching

Figures 5.10, 5.11, 5.12, 5.13 show the performance of the two Caching architectures. In addition, we also include the top performers from other architecture types (Dynamic Hot-DST, Interval-Driven Hot-DST (eighty-second), and Plain or Dynamic Cold-DST depending on the workload) so that we can perform a side-by-side comparison of Caching and DST architectures. Several interesting observations can be made from these figures.

First, as the Web Server workload with Zipf locality reveals (Figure 5.13), all hybrid configurations perform significantly better than the HDD-only case at all SSD sizes, thus, proving the effectiveness of hybrid architectures.

Second, On-demand Caching consistently outperforms Interval-driven Caching under all workloads. The same reasoning behind Dynamic Hot-DST being faster than its Interval-driven counterpart applies here as well—Interval-driven Caching suffers from the same thrashing issues as its DST counterpart.

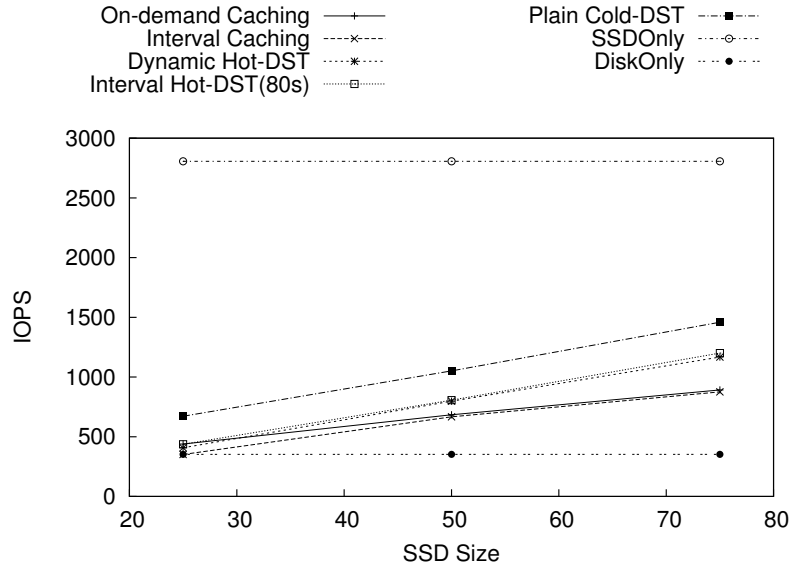


Figure 5.11: IOPS delivered under FileBench's File Server workload by Caching and DST architectures.

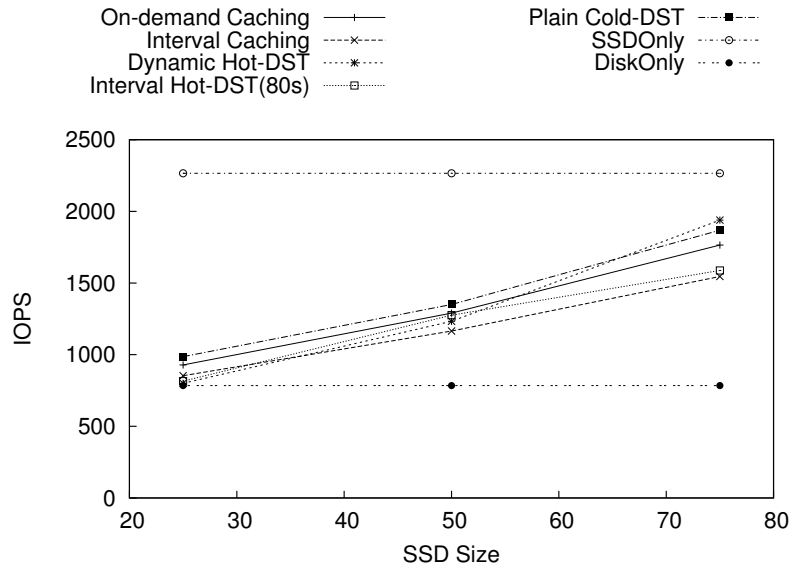


Figure 5.12: IOPS delivered under FileBench's Web Server workload by Caching and DST architectures.

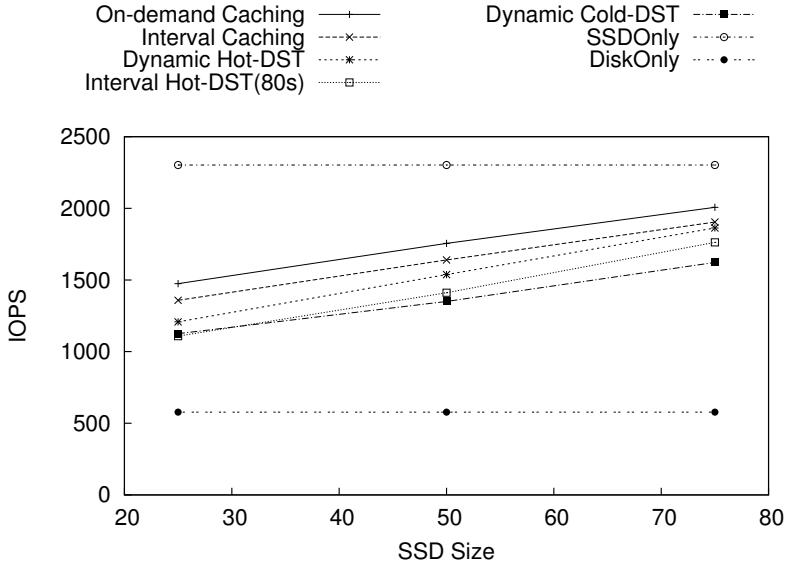


Figure 5.13: IOPS delivered under FileBench's Web Server (with Zipf) workload by Caching and DST architectures.

Third, Postmark and File Server workloads (Figures 5.10, 5.11) reveal the impact of an important design parameter—the presence or absence of a data copy. As we mentioned earlier, the DST and Caching implementations are identical in all aspects except for the fact that Caching implementation copies files while the DST implementation migrates them. Since we implemented Write-through-update Caching, any writes to these cached copies are also written through to their primary disk replica. Since Postmark and File Server workloads consist of append operations, all Caching architectures suffer due to the necessity to keep the two copies in sync. Thus, they perform consistently worse than their DST counterparts. This shows how Hot-DST architectures are preferable over their Caching counterparts under workloads with a low read:write ratio.

Fourth, the Web Server workload (Figures 5.12, 5.13) reveals a drawback inherent to DST. Unlike Postmark and File Server, we see that the Caching architectures outperform their DST counterparts at low SSD sizes. As we mentioned earlier, under the Web Server workload, the only write operations issued are those that append data to the log file. Since our implementation pins the log file to the HDD tier, SSDs are used for serving only reads. As a result, Caching architectures incur no consistency-related overhead. Furthermore, SSD tier cleaning under Caching architectures involves invalidating the cached copy by just deleting it. DST architectures, on the other hand, have to migrate "cold" data back to the HDD tier incurring an additional overhead. This shows how Caching architec-

tures are preferable over their Hot-DST counterparts under workloads with a high read:write ratio.

Fifth, Cold-DST architectures meet or exceed the performance achieved by other architectures under all workloads except the Web Server workload with locality (Figure 5.13). We believe that this oddity is in large part due to the preallocation phase. As we explained earlier, FileBench first preallocates files before starting the workload generators. As Cold-DST architectures allocate files on the SSD tier, they start operating with a full SSD tier. Hot-DST and Caching architectures, on the other hand, start with a near-empty SSD tier as they perform initial allocation on the HDD tier. Thus, "hot" file migration under Hot-DST and Caching architectures incurs no cleaning overhead until the SSD tier gets full. However, dynamic migration under Cold-DST architectures incurs cleaning overhead from the very beginning causing a noticeable performance drop.

5.5.5 Mixed Workloads and Hybrid Architectures

In order to analyze the effect of storage virtualization on the performance of various hybrid architectures, we wrote a FileBench workload model that blends File Server and Web Server workloads into a single stream. We used the same configuration parameters as the individual workloads. In addition, we also preallocated a dummy fileset to simulate shifting locality by flushing valid data off the SSD.

Type-aware DST

In order to understand if pairing workloads with ideal architectures is better than adopting a "one-architecture-for-all" approach, we modified the Loris stack to support *Type-aware DST*. Our Type-aware DST implementation associates a tiering policy with each file volume (a rooted hierarchy of files and directories). We created two file volumes, one per workload, and tag volumes with policies that directed the logical layer to pair On-demand Caching with the Web Server workload and Dynamic Hot-DST with the File Server workload. Thus, our Type-aware DST implementation caches or migrates "hot" files depending on whether they belong to the Web Server or File Server volume.

Figure 5.14 shows the performance of individual Caching and DST architectures side-by-side with our Type-aware DST architecture under the mixed workload. There are three important observations to be made. First, Dynamic Cold-DST meets the performance of type-aware tiering at low SSD sizes, and exceeds it at higher sizes. Contrasting this with the performance of Dynamic Cold-DST under just the Web Server workload (Figure 5.8), we clearly see that the performance improvement achieved by allocating new files in the SSD tier overshadows the adverse effect of dynamic migration.

Second, the Caching configuration suffers under the mixed workload. This

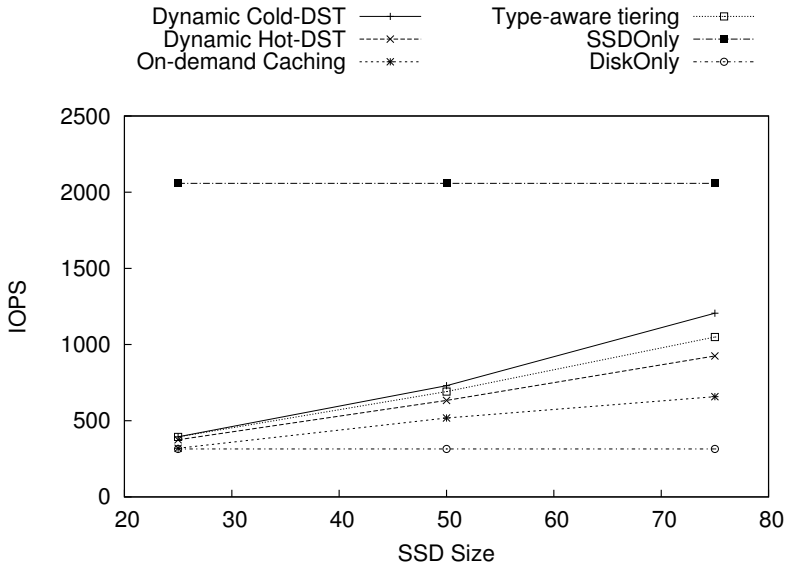


Figure 5.14: IOPS delivered under FileBench's mixed workload by Caching and DST architectures.

can be attributed to the synchronization overhead caused by append operations in the File Server workload.

Third, the Type-aware DST outperforms both individual architectures at all SSD sizes as it possesses the advantages of both Caching and Dynamic Hot-DST architectures without any of their disadvantages. By migrating files created by the File Server workload, and caching files created by the Web Server workload, Type-aware DST reduces the cleaning overhead without incurring the expensive synchronization overhead. This illustrates the benefit of pairing workloads with their ideal architectures. The Type-aware DST architecture we implemented is only one of many possible alternatives. For instance, one could also pair Cold-DST architecture with the File Server workload and Dynamic Hot-DST architecture with the Web Server workload. We intend to implement such architectures and evaluate them using file system traces as a part of future work.

5.6 Discussion

Based on our experience designing and evaluating various hybrid architectures, we will now present a few open research problems that need to be solved in order to be able to design efficient hybrid storage architectures.

5.6.1 Analyzing Cold-DST

While Hot-DST systems have received a lot of attention over the past years, we believe Cold-DST architectures have been ignored due to two main reasons, namely, poor write performance, and inferior reliability. The design of early hybrid systems that used first generation SSDs focused on improving the overall system performance by pairing read-only data with SSD tier and write-only data with the HDD tier. Researchers have pointed out that the limited lifetime (erasure cycles) of NAND-flash-based SSDs must be considered as a critical factor in the design of hybrid storage systems, and have even proposed using HDD-based logging to improve the longevity of SSDs [89]. As Cold-DST architectures write significantly more data to the SSD tier than other architectures, they will most certainly wear out these SSDs faster.

However, unlike first generation SSDs, which suffered from poor random write performance due to inefficient Flash Translation Layer (FTL) designs (among several other reasons), modern SSDs have exceptionally high write performance, sometimes even exceeding read performance. Similarly, modern enterprise-grade, SLC flash-based SSDs have reasonably high reliability. For instance, OCZ Vertex2 EX SLC SSD has an MTBF rating of 10 million hours.

In light of these recent changes in the storage hardware landscape, modern DST systems (like Hystor [19]) have started allocating dedicated write-back areas in SSDs to improve write performance. Cold-DST architectures are capable of meeting the performance offered by Write-back Caching without any of the synchronization-related performance issues. However, there are several important design factors that require further research. Should a Cold-DST implementation partition the SSD space into read and write areas, and if so, can it dynamically determine partition sizes? Recent analysis of network file system traces indicate that over 90% of newly created files are opened less than five times [61]. Can we utilize SSD parallelism to perform aggressive cold migration without affecting foreground accesses under such workloads?

5.6.2 Other hybrid architectures

In addition to DST and Caching architectures, there are several other ways SSDs could be integrated into the storage stack. For instance, SSDs could be used as dedicated data stores for housing specific data types. One such example is using SSDs for exclusively storing file system metadata, executables and shared libraries, as suggested by the Conquest file system [104]. Researchers have shown how MEMS-based storage can be used in several capacities to accelerate performance of disk arrays [100]. Similarly, SSDs could also be used in heterogeneous disk arrays to eliminate redundancy-related performance bottlenecks. To our knowledge, such configurations have not been compared side-by-side with

DST or Caching architectures, and such a systematic study would help determine the best possible way to integrate SSDs into the storage stack.

5.6.3 Caching vs Tiering Algorithms

Earlier in this paper, we showed how inverse bitmaps, a "hot" data identification mechanism originally used to implement a DST system, can also be used to implement effective Caching architectures. The cross-architecture applicability of several data collection algorithms raises several research questions like 1) how effective are second-level buffer cache management algorithms when used to implement DST architectures?, 2) does the relative performance of various architectures remain unaffected across different data collection algorithms?

We intend to use the Loris stack to implement several hybrid architectures and answer these research questions as a part of future work.

5.7 Conclusion

We showed how our plugin-based extensions to the Loris stack transform it into a framework for implementing hybrid storage solutions. Using the Loris framework, we illustrated the effectiveness of DST and Caching by showing how these hybrid architectures can outperform a disk-only configuration even with SSD sizes covering as little as 25% of the working set. Based on our evaluation, we offered several insights into interactions between architecture-specific design alternatives and workload parameters. We also discussed the ramifications of our work by highlighting a few areas that deserve more attention from storage researchers.

Chapter 6

File-Level, Host-Side Flash Caching with Loris

Abstract

As enterprises shift from using direct-attached storage to network-based storage for housing primary data, flash-based, host-side caching has gained momentum as the primary latency reduction technique. In this paper, we make the case for integration of flash caching algorithms at the file level, as opposed to the conventional block-level integration. In doing so, we will show how our extensions to Loris, a reliable, file-oriented storage stack, transform it into a framework for designing layout-independent, file-level caching systems. Using our Loris prototype, we demonstrate the effectiveness of Loris-based, file-level flash caching systems over their block-level counterparts, and investigate the effect of various write and allocation policies on the overall performance.

6.1 Introduction

Over the past few years, many enterprises have shifted from using direct-attached storage to network-based storage for housing primary data. By providing shared access to a large volume of data and by consolidating all storage resources at a single spot, network-based storage improves scalability and availability significantly. The storage industry has also witnessed an equally phenomenal increase in the adoption of flash-based solid state storage. While flash can be used in several capacities (data/metadata caches, primary storage devices, etcetera) in a networked storage server, recent research has shown that using flash at the host rather than server side has several advantages [18; 82]. First, a hit on the host-side flash cache can be serviced immediately without an expensive network access. Second, by filtering requests, a host-side cache significantly reduces the number of requests that need to be serviced by the storage server. By eliminating bursty traffic, host-side caching enables storage servers to be provisioned for average I/O volumes, rather than peak volumes, thereby reducing capital expenses.

Figure 6.1 shows the architecture and components involved in a typical host-side caching implementation. As shown in the figure, storage resources consolidated at the server side are exported to the host side using a Storage Area Network (SAN) protocol like iSCSI. File systems at the host, which traditionally managed direct-attached storage devices, are now used to manage remote storage volumes. Several systems integrate caching into the storage stack below the file system and above the iSCSI client, thereby retaining backward compatibility with existing file systems.

Block-level caching systems can be classified into two types depending on their write policy, namely, *write through* and *write back*. A write-through cache issues writes to both the networked primary storage and the local flash cache in the order in which they were received and waits for these writes to complete before passing back an acknowledgement to the file system. A write-back cache, on the other hand, acknowledges writes as complete as soon as they are serviced by the local flash cache. The networked primary storage is updated asynchronously in background. Under write-intensive workloads, write-back caching is guaranteed to improve performance as it converts high-latency, foreground writes into asynchronous, low-latency background writes. However, as a block-level, write-back cache intercepts and caches file system requests, it absorbs writes to both metadata and data blocks alike, thereby causing several problems as it silently changes the file system-enforced data ordering.

The first issue is the impact on the correctness of several server-side administrative operations such as backup, snapshoting and cloning. For instance, a snapshot initiated at an inopportune time might capture an inconsistent image of the data volume. Thus, backups made off this snapshot would not help in disaster recovery, as the host file system might not be able to fix the inconsistency in the

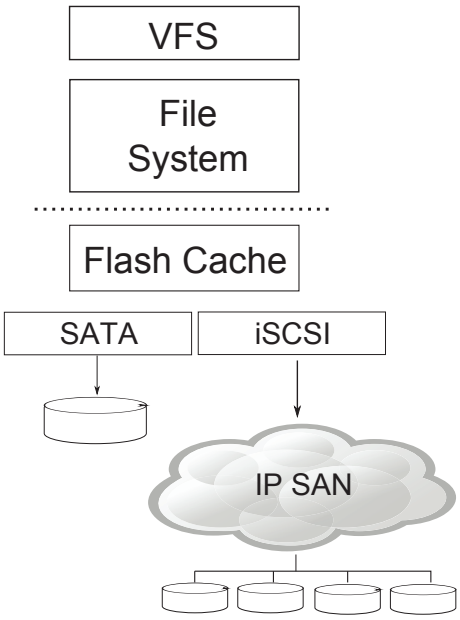


Figure 6.1: Traditional host-side flash caching architecture. The figure shows pooled storage resources at the server side exported to the host side over an IP-based Storage Area Network. On the host side, the dotted lines demarcate file-aware layers from those that are not. Thus, the flash cache is managed by a file-unaware, block-level cache driver that uses the SATA and iSCSI subsystems to communicate with the local cache device and remote primary storage. The semantically-aware file system remains oblivious to the usage of a cache device.

restored snapshot. Second, a loss of file system metadata due to an SSD failure could have a negative impact on availability as it can cause substantial data loss. Third, almost all state-of-the-art enterprise storage systems adopt the "release consistency" model [18] when multiple storage clients access the same storage volume. Under this model, a volume is shared across clients in a serial fashion with only one client maintaining exclusive access at any time. When block-level write-back caching is used in such a setting, the failure of one client can render the network storage inconsistent or, in the worst case, unusable by any other client. In light of these issues, it is not surprising that several write-back caching implementations even make explicit disclaimers warning administrators about storage-level inconsistencies after a cache failure [21].

6.1.1 Consistent, Block-Level Write-Back Caching

To solve these issues caused by unordered write-back policy, two solutions were proposed recently [54]. The first technique, referred to as *Ordered Write Back*, is

based on the simple idea that consistency at the networked storage system can be maintained by evicting blocks in the same order in which they were received by the cache. The intuition behind this idea is the fact that file systems already maintain a consistent disk image by enforcing ordering of write requests (for example, journaled writes must precede actual data writes). However, this approach has several bottlenecks that impose a significant performance penalty. For instance, the cache must keep track of dependencies between data blocks – an operation with non-trivial compute and memory requirements. It must also preserve and write back all dirty copies of the same block, thereby wasting cache space and network bandwidth.

To solve problems with Ordered Write Back, *Journaled Write Back* has been proposed. The idea behind this approach is to use a host-side, persistent journal, in concert with a journal on the server side, to bundle file system updates into transactions that are checkpointed asynchronously to remote storage in the background. Thus, the Journaled Write Back approach enables consistent, block-level, write-back caching at the host side only when used in concert with networked storage servers that provide a atomic-group-write interface. In addition, under certain configurations, this approach would suffer from performance issues due to redundant use of journaling by both the file system and block-level cache to protect the same data and metadata blocks. Recently, researchers have shown how such redundant journaling, albeit in a different context (SQLite database and ext4 file system), deteriorates application performance significantly in the Android stack [47].

6.1.2 Filesystem-Based Caching

Given that all modern file systems use techniques like journaling and shadow copying to ensure metadata consistency across reboots, the natural alternative to integrating caching at the block level is to modify existing file systems to be cache aware. However, such an integration has one major issue – its lack of portability. A caching algorithm integrated into a file system is restricted to work only within the scope of that file system. This lack of portability would only be an inconvenience rather than a show stopper if device heterogeneity were nonexistent.

Heterogeneity exists both within and across device families. New devices, with interfaces different from the traditional block-based read/write interface, are emerging in the storage market. For instance, some flash devices and Storage Class Memory devices are byte accessible, while Object-based storage devices, on the other hand, work with objects rather than blocks. Integrating these devices into the storage stack requires building custom file systems, and hence reimplementing the caching algorithm, for each device family.

Similarly, different SSDs, sometimes even from the same vendor, have different performance characteristics. For instance, Intel X25-V SSD design makes a price/performance trade-off, as it sacrifices sequential read/write throughput by

reducing the number of channels populated with NAND flash. Intel X25-M, on the other hand, has equally impressive random and sequential read/write performance figures. Achieving optimal performance in such cases requires pairing devices with their ideal layout algorithms. For instance, a log-structured layout might be best suited for an Intel X25-M, while it could deteriorate performance when used with X25-V. This heterogeneity in layout management forces one to reimplement caching algorithms not just across device families, but also for each new layout algorithm within device families.

6.1.3 Our Contributions

Solving the heterogeneity issues faced by file system-based caching solutions requires decoupling flash-cache management from layout management. In prior work, we proposed Loris [9], a fresh redesign of the storage stack that implements layout-independent, file-level RAID algorithms. In this paper, we present our design extensions to Loris that transform it into a framework for implementing layout-independent caching solutions. In doing so, we make three major contributions to state of the art.

First, in contrast to traditional approaches, we make the case for integrating caching algorithms at a different level in the storage stack (Section 6.2). With the new integration, caching algorithms work at a higher level of abstraction by managing files rather than disk blocks. As we will see later in this paper, one of the major challenges with such an integration is implementing efficient subfile caching. Thus, our second contribution is the Loris-based subfile caching framework that can be used by any caching algorithm to map each logical file block to a different storage target (Section 6.3). Our third contribution is a thorough comparative evaluation of our Loris prototype with a block-level solution to prove the effectiveness of our approach against a traditional block-level cache, and to understand the impact of caching policies on overall performance (Section 6.4).

File-level caching has been implemented earlier in the context of distributed file systems like AFS [46] and Coda [52]. In these systems, the client implementation runs as a user-space application and uses the local file system to perform coarse-grained, whole-file caching of application data (not system metadata) stored in a networked file store. We, on the other hand, integrate flash-caching algorithms directly into the local storage stack and show how a file-level (but not whole file) integration of caching algorithms can be used to implement unified, block-granular caching of both application data and system metadata, without any of the consistency issues or performance overheads of the traditional block-level integration.

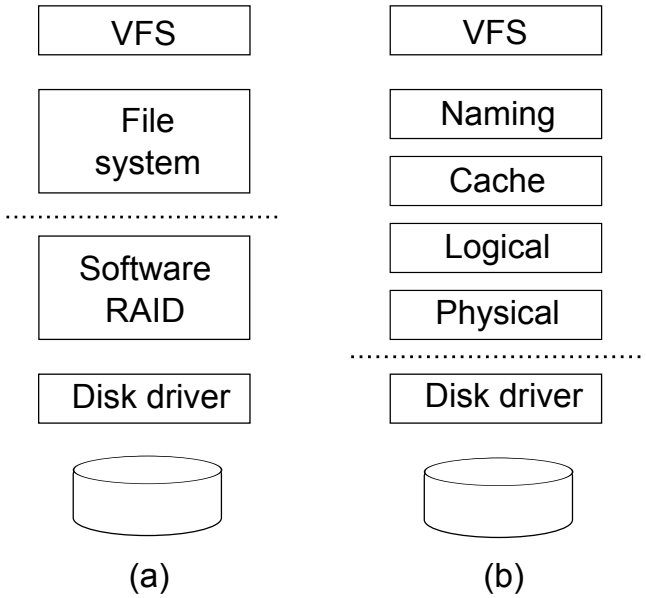


Figure 6.2: The figure depicts (a) the arrangement of layers in the traditional stack, and (b) the new layering in Loris. The layers above the dotted line are file aware; the layers below are not.

6.2 The Case For File-level Host-Side Caching With Loris

In this section, we will first provide a quick overview of the Loris storage stack. Following this, we will show how Loris makes layout-independent integration of flash caching possible, and we will make the case for such an integration by describing its advantages over file system-based and block-level approaches.

6.2.1 Loris - Background

Loris is made up of four layers as shown in Figure 6.2. The interface between these layers is a standardized file interface consisting of operations such as *create*, *delete*, *read*, *write*, and *truncate*. Every Loris file is uniquely identified using a <volume identifier, file identifier> pair. Each Loris file belongs to a file volume, which is a rooted collection of files and directories. Each Loris file is also associated with several *attributes*, and the interface supports two attribute manipulation operations—*getattribute* and *setattribute*. Attributes enable information sharing between layers, and are also used to store out-of-band file metadata. We will now briefly outline the responsibilities of each layer in a bottom-up fashion.

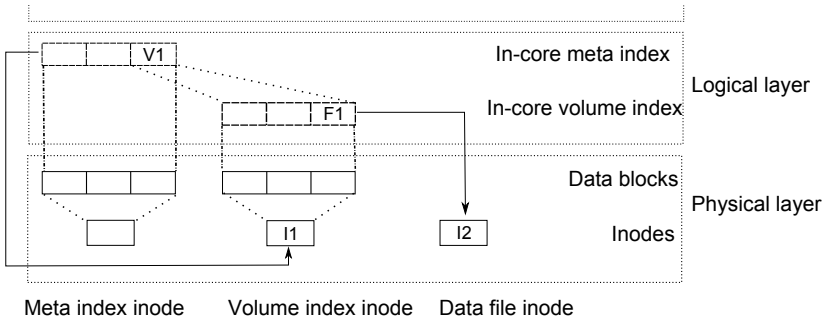


Figure 6.3: The figure shows the relationship between meta index and volume index. The figure shows the meta index file containing the file volume metadata entry for volume V1, which could be <V1, REGULARVOL, volume index configuration=<raidlevel=1, stripesize=N/A, physicalfiles=<D1:I1>>. Thus, inode I1 in physical module D1 is used to store the volume index file data (an array of logical file configuration entries) for file volume V1. The logical file configuration entry for file <V1, F1> could be <raidlevel=1, stripesize=INVALID, physicalfiles=<D1:I2>>. Thus, inode I2 in physical module D1 is used to store file F1's data.

Physical Layer

The physical layer exports a physical file abstraction to the logical layer. A physical file is a stream of bytes that can be read or written at any random offset. Thus, details such as the device interface and on-disk layout are abstracted away by the physical layer.

Each physical layer implementation is tasked with providing 1) device-specific layout schemes for persistent storage of files data/attributes, and 2) end-to-end data verification using parental checksumming. Each storage device is managed by a separate instance of the physical layer, and we call each instance a *physical module*. Our current physical layer prototype is based on the traditional UNIX file system layout. Each physical file is represented by an inode. Each inode contains enough space to store the file's Loris attributes, seven direct data block pointers, and one single, double and triple indirect block pointer. Free blocks/inodes are tracked using block/inode bitmaps. Although our physical layer implements parental checksumming of all data and metadata, we will omit the details as we do not use it in our evaluation.

Logical Layer

The logical layer exports a *logical file* abstraction to the cache layer. A logical file is a virtualized file that appears to be a single, flat file to the cache layer. Details such as the physical files that constitute a logical file, the RAID levels

used, etc. are confined within the logical layer. The logical layer works with physical files to provide both device and file management functionalities. It is made up of two sublayers, namely the file pool sublayer at the bottom, and the volume management sublayer at the top.

In prior work, we introduced a new Loris-based storage model called File Pooling, that simplifies management of storage devices [10]. File pools simplify storage administration and enable thin provisioning of file volumes. The file pool sublayer maintains data structures necessary for tracking device memberships in file pools, and provides device management operations for online addition, removal and hot swapping of devices.

The volume management sublayer supports file volume virtualization. As we mentioned earlier, each logical file belongs to a file volume. Each file volume is physically represented by a *volume index file* which is created at file volume creation time. This file stores *file configuration information* entries for all files belonging to its volume. This configuration information consists of 1) RAID level used, 2) stripe size used (for certain RAID levels), and 3) list of physical files that store the logical file's data.

Similar to the way the volume index file tracks the membership of files in file volumes, file volumes themselves are tracked by the *meta index file*. This file contains *file volume metadata* entries, one per volume, that record: 1) the number of files in that volume, 2) tiering/caching policy used, and 3) physical file(s) that store the volume index data among other details. Figure 6.3 describes the relationship between these two data structures with an example.

Cache and Naming Layers

The cache layer provides in-core caching of data pages. Our prototype cache layer implements the LRU cache replacement algorithm.

The naming layer acts as the interface layer. Our prototype naming layer implements the traditional POSIX interface by translating POSIX files and attributes into their Loris counterparts. It implements the directory abstraction by using Loris files to store directory entries. All POSIX semantics are confined to the naming layer. However, the naming layer uses the attribute infrastructure to help the other layers discern Loris metadata from application data. For instance, as far as the logical layer is concerned, directories are just regular files with special attributes that mark them as important. It uses this information to mirror directories on all physical layers for improving availability.

Crash Recovery in Loris

Similar to other systems, Loris also uses snapshot-based recovery to maintain metadata consistency across system failures. Due to lack of space, we will just

present an overview here and we would like to direct the reader to [102] for further details. During normal operation, after every preconfigured time interval, Loris takes a system-wide snapshot. During this operation, all Loris layers flush out any dirty data and metadata that is yet to be written. Then, the logical layer asks each physical module to take a snapshot of all metadata (both physical module's layout-specific metadata and those belonging to the other Loris layers) and tag the snapshot with a common timestamp. It is important to note here that only metadata, not application data, is snapshotted and the physical layer can distinguish metadata from data using attributes as we mentioned earlier. Thus, each global metadata snapshot can be identified using a single timestamp across all physical modules.

After a system failure, logical layer probes all physical modules for their latest timestamp. If the system had shutdown gracefully, all physical modules would return back the same timestamp. A disparity in timestamp indicates an unclean shutdown, upon which the logical layer instructs all physical modules to roll back metadata to the latest common timestamp. Thus, in a nutshell, the task of providing consistency in Loris is divided between the logical and physical layers. Each physical module is tasked with implementing some form of metadata snapshoting. The logical layer works with physical module snapshots and coordinates recovery to a globally consistent snapshot after a system failure.

6.2.2 File-level Host-side Caching With Loris

Comparing Loris with the traditional stack (Figures 6.1, 6.2), one can observe two things. First, the file system, which is a monolithic module in the traditional stack, has been decomposed into naming, cache, and physical layers in the Loris stack. Second, as the dotted line indicates, all Loris layers operate at the file level in contrast to the traditional stack, where RAID and caching algorithms operate at the block level. It is because of these two fundamental design differences that Loris enables a new level (the logical layer) at which flash caching can be integrated.

Figure 6.4 shows how Loris can be used as a host-side cache. Loris runs on the host machine as the primary file system and manages both the local SSD and the remote iSCSI volume. Physical layer implementations customized for SSD and iSCSI storage map device blocks to Loris physical files. The caching logic (allocation and replacement algorithms), however, is implemented at the file level, in the logical layer, in contrast to the traditional caching design where it is integrated at the block level. This integration possesses all the advantages of a file system-based approach without any of its disadvantages.

First, the file-level implementation of caching makes it device or storage interface agnostic. Switching to a new type of caching device (like MEMS or Object-based Storage (OSD) instead of SSD) requires just implementing corresponding physical modules. Thus, file-level caching obviates translation layers as there is

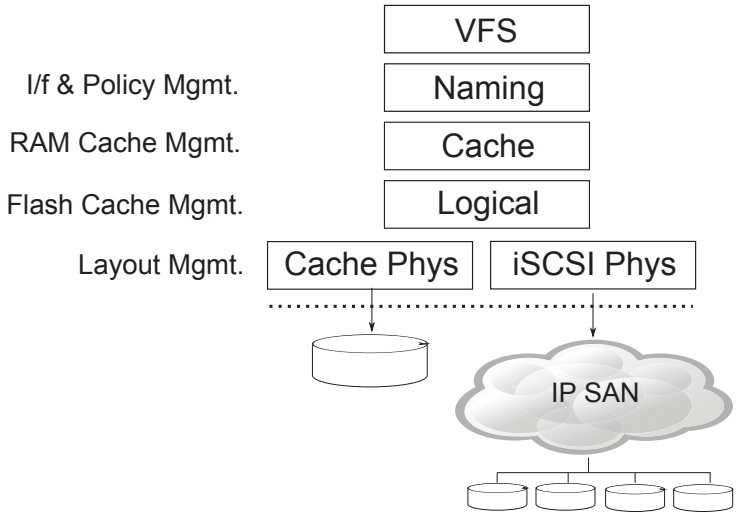


Figure 6.4: Host-side flash caching with the Loris stack. The figure shows the roles and responsibilities of each layer when the Loris stack is used as a host-side caching solution. Contrasting this with Figure 6.1, one can see that the local flash cache is managed by the file-aware logical layer.

no necessity to map any device interface to a generic block interface. In the absence of such abstractions, device-specific physical layer implementations can implement highly-customized optimizations that exploit advantages specific to each device family. For instance, one could implement a short-circuit-shadow-paging-based physical layer for a PCM device [20], or a physical layer that exploits the virtualized-flash-storage abstraction offered by modern PCI Express SSDs [48], without affecting the caching implementation. Later in this paper, to show the benefit of interface-agnostic flash caching, we will describe our implementation of a simplified NFS-client-like physical layer that enables to usage of any networked, file-based remote storage system as primary data store.

During normal operation, the logical layer treats all physical modules (local and remote) alike and establishes global metadata checkpoints across them. Irrespective of where they are stored, all metadata updated between two checkpoints get persisted as a part of the next global checkpoint, or reverted during recovery after a crash, as a single atomic unit. Thus, the second benefit is that any Loris-based caching implementation can recover from OS crashes and power failures on the host side using Loris' built-in consistency mechanism without any additional effort.

We would like to explicitly mention here that these consistency-enforcing metadata checkpoints are created and maintained by the physical layer implementations and thus, have no influence on logical layer-resident, flash-cache man-

agement algorithms that perform caching of application data. Also, these checkpoints are different from administrator-triggered file volume snapshoting of all data and metadata. While Loris is capable of supporting such snapshoting, and while the interaction between snapshoting and caching certainly requires special attention [18], our focus in this paper is on using Loris as a host-side cache with any server-side NAS or SAN appliance. In this scenario, administrators typically use server-side (not client-side) snapshoting facilities for performing various administrative operations. We intend to integrate host-side snapshoting with the caching framework described in this paper as a part of future research which involves investigating the utility of Loris as a hypervisor flash cache in virtualized data center (Section 6.5).

Third, by being file aware, Loris can use different caching policies for different file types. For instance, Loris could associate all metadata with the write-through policy. Thus, even if the user specifies a write-back policy for all application data, writes by the naming layer to directory files and by the logical layer to volume index file will be written through immediately to the networked storage server. By having all metadata written through immediately, Loris can recover from all host side failures. For instance, a failure of the SSD on the host side would only result in application data loss and never renders the networked primary storage unusable. Inconsistencies between data and metadata caused by an SSD failure can be easily identified by the logical layer and propagated to the application on demand, thereby providing high availability. Thus, Loris provides a framework for implementing persistent, file-level write-back caching systems that do not suffer from any of consistency issues that plague the block-level integration.

Fourth, as the caching algorithms are plugin-based, Loris can easily pair workloads with ideal caching algorithms in contrast to even modern, state-of-the-art cache-aware file systems that adopt a single, one-size-fits-all approach to flash caching. This flexibility is especially important in modern data centers where server consolidation forces a single host-side caching system to service requests from disparate workloads with different RPOs. For instance, ZFS [6] uses SSDs that can sustain high random IOPS for caching read-only data and SSDs with high sequential write throughput for storing the ZFS journal (ZIL) [62] irrespective of application workload. Loris, on the other hand, could pair workloads with write-back or write-through caching depending on their RPO.

Although Loris provides a convenient framework for implementing host-side caching systems, the whole-file nature of mapping maintained by the logical layer makes it impossible to implement fine-grained, subfile caching, as caching a single file block requires caching the entire file. Even worse if the fact that such subfile caching is mandatory for implementing write-back caching, where files can be arbitrarily written/updated in small chunks. Thus, the logical layer must be completely redesigned to support fine-grained, subfile caching.

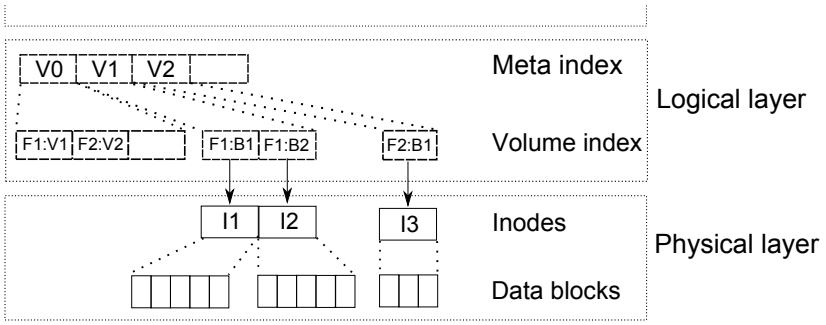


Figure 6.5: File-as-a-volume subfile mapping approach: The figure shows blocks F1:B1, F1:B2 of file < V0, F1>, and block F2:B1 of file < V0, F2> are mapped to physical files I1, I2 (for F1), and I3 (for F2).

6.3 Loris-based Host-side Cache: Architecture

Our new logical layer consists of three plugin-based sublayers: 1) the volume management sublayer with extended support for subfile caching, 2) the cache management sublayer that manages the flash-based host-side cache, and 3) the file pool sublayer which provides device management and RAID services. Each of these sublayers has a well-defined interface, similar to the Loris interface, and can be replaced without changing the other sublayers. As device management and RAID algorithms are out of the scope of this paper, we will now describe in detail the first two sublayers.

6.3.1 Volume Management Sublayer: Subfile Mapping

As we mentioned earlier, our original logical layer maps each logical file to one or more physical files. However, implementing subfile caching requires mapping each logical file block (not the whole file) to one or more physical files. In addition, we also need to maintain metadata that identifies the block cached in the SSD as clean or dirty; the action taken during cache eviction varies depending on the block state (clean data can be just discarded whereas dirty data must be written back to primary storage).

As the relationship between files and blocks is very similar to the relationship between file volumes and files, we initially implemented this indirection by recursively extending the file volume abstraction. When a logical file was created, an entry was allocated for it in its parent file volume's volume index as before. In addition, we created a new volume, whose logical configuration information entries record the logical block–physical file(s) mapping for each block as shown in Figure 6.5. Thus, by treating each logical file as a file volume, we were able to extend the existing abstraction to support subfile mapping with minimal effort.

However, preliminary evaluation revealed that the overhead caused by metadata allocation and lookup was a significant source of performance degradation. With the aforementioned subfile mapping, we were storing physical file information for each logical block. Each physical file is represented by a <moduleid, inode number> pair, which is encoded using four bytes in our current implementation. Assuming an block size of 4-KB, and assuming that all blocks are cached on the SSD, a 4-GB file would require 8-MB (eight bytes per block, four for HDD physical file and four for the SSD one).

To eliminate the metadata overhead, we increased the mapping granularity from a block to an extent - a logically contiguous group of blocks. While this did improve performance significantly, it still suffered from two problems. First, as each extent was stored as a separate physical file, at small extent sizes, most read/write requests would need to be serviced by reading/writing multiple physical files. Despite the fact that our prototype exploits the Native Command Queuing (NCQ) capability of both HDD and SSD by queuing these reads/writes in parallel, we found that splitting a single, large read/write request into several constituent extent-sized requests had a significant performance impact. Second, extent-granular mapping complicates write-back caching as caching algorithms and staging/eviction of data must be extent aligned. In addition, writes misses force the entire extent to be read from the disk, if not already cached, causing performance deterioration at large extent sizes. Thus, we abandoned the "file-as-a-file-volume" approach and adopted a newer one.

Our new approach is based on the insight that rather than storing each logical block in a separate physical file, we could use a single physical file to pack related blocks. In other words, each logical file could be associated with two physical files, one on the SSD, and the other on the iSCSI volume. By doing so, for each logical block, we would need to record whether 1) it has been cached on the SSD physical file, and 2) if the SSD copy is dirty. We could easily accomplish this with just two bits per block. Thus, in contrast to the previous approach, a 4-GB file can be encoded using 128-KB for caching status bits, 128-KB for the dirty bits, and 8 bytes for the physical file information. Thus, we reduce the metadata footprint by a factor of 32 (8 bytes to 2 bits per entry). In addition, a large request spanning multiple blocks would now translate to a single physical file read (assuming that the corresponding blocks are all sequential and collocated in the same device).

In our current prototype, we implemented this new mapping by modifying the file configuration information stored by the volume management sublayer. The new configuration information contains 1) physical file information for the primary file in the iSCSI physical layer, 2) physical file information for the cached SSD file (if cached), 3) a 32-byte block status bitmap, and 4) a 32-byte dirty bitmap. In order to implement fine-grained caching, we use a block size equal to the OS page size of 4-KB. Thus, the block information for all files up to 1-MB resides entirely within its logical configuration information. When a file grows over

1-MB, we dynamically create a new physical file and use it to store both bitmap blocks.

6.3.2 Cache Management Sublayer

There are two main design parameters that influence the operation of a host-side caching system. The first parameter is the write policy (write through or write back, as we mentioned in Section 6.1). The second parameter is the allocation policy that controls when data is admitted into the cache. Based on this policy, caches can be classified as *write-allocate* and *write-no-allocate* (also known as *write-around*). As the names imply, the former policy admits data on write misses while the latter does not. As we wanted to systematically study the effect of each parameter on overall performance, we implemented all four possible host-side caching alternatives as a separate cache management plugin.

Although these plugins differ with respect to write and allocation policies, they all share two things in common. First, they all admit data into the cache as a side effect of a read miss. Second, they all use the LRU replacement algorithm to manage the flash cache. In our current prototype, the algorithm is implemented as a separate component independent of all plugins. It maintains an in-memory list of entries, on per logical file block, in LRU order. As this list does not contain information about the physical location of logical file blocks (which is recorded in volume index entries and protected using Loris' consistency mechanism), it can be maintained entirely in memory as a power failure or system crash would result only in the loss of recency information. In addition, as each list entry only needs to store the logical file id and offset for currently cached blocks, it can easily scale to large cache sizes.

6.3.3 Physical Layer Support For Subfile Caching

The aforementioned changes to the logical layer make fine-grained cache admission possible. However, as caching algorithms work at a page granularity, they also require the capability to evict individual pages for freeing up cache space. As caching algorithms operate at the logical layer, there were two ways a Loris-based cache implementation could free up space, namely, deleting a physical file or truncating it. We found both these operations to be too coarse grained and inefficient as it is impossible to implement fine-grained individual page/block evictions using either of these methods.

To solve these problems, we added a new *rdelete*(range delete) operation to the physical layer API. Only those physical modules that will be used to manage cache devices need to support this operation. The logical layer uses rdelete to free arbitrary data ranges in physical files. When space is needed in the flash cache for accommodating new data, the LRU algorithm is invoked to find the longest

sequence of logically contiguous file blocks (blocks belonging to the same logical file at consecutive offsets). The logical layer then issues an rdelete call to the physical module that manages the flash cache which, in turn, frees those data blocks and associated indirects, effectively creating holes in that physical file.

6.3.4 Network File Store

Although our traditional Loris physical layer can be used over an iSCSI volume, we wanted to prove the utility of layout-independent, file-level flash caching. So, we implemented a new physical module that can communicate with any file server that support four basic calls to create, delete, read from and write to a file. This network file client is essentially a simplified NFS client that maps calls from the Loris interface to the limited file server interface. To make our client portable across file servers, we also implemented support for attribute handling at the client side. Each client maintains a special file, which is created during startup, in which it stores the attributes for all Loris files. Thus, while read/write calls for Loris files get mapped onto file server read/write calls, getattr and setattr operations directed at the network client are implemented reading/writing into this special file.

6.4 Evaluation

In this section, we will first describe the hardware/software setup and benchmarking tools following which we will present our comparative analysis of the four Loris-based cache management architectures to understand the impact of each policy on overall performance.

6.4.1 Setup

The client machine we used in all tests was an Intel Core 2 Duo E8600 PC, with 4-GB RAM. We used a OCZ Vertex3 Max IOPS SSD as our host-side flash cache. Our networked file server is an Intel Core2Duo E8600 PC, with 4-GB of RAM and a 500-GB 7200 RPM Western Digital Caviar Blue (WD5000AAKS) SATA drive. In all experiments, we used the first 8-GB of both SSD and HDD to house all data.

Both client and server machines run MINIX 3, a microkernel, multiserver operating system [97]. Loris runs on the client machine and manages the flash cache while we use the MINIX 3 File System on the server machine as our network file store. We deliberately configured Loris (on the client side) to run with only 64-MB RAM cache (cache layer) to ensure that our flash-caching subsystem (in Logical and Physical layers) is stressed thoroughly. As we vary the flash cache size in some experiments, we will report the actual SSD size used while describing the results of each experiment.

Benchmark	WB	WB-WA	WT	WT-WA
Postmark (secs)	458	2084	1348	2304
File Server (IOPS)	1046	231	255	199
Web Server (IOPS)	3423	2856	3472	2859

Table 6.1: Execution time (in seconds) and IOPS achieved by write-back (WB) and write-through (WT) cache plugins, with and without write-around (WA) allocation, under various benchmarks, at a cache size of 1536-MB.

Although we implemented all the features we described in Section 6.3 in our prototype, we will focus only on the performance aspect of Loris-based file-level caching in this paper.

6.4.2 Benchmarks and Workload Generators

We used Postmark and FileBench to generate four different classes of server workloads for our comparative evaluation. Postmark is a widely used, configurable file system benchmark that simulates a mail server workload. We configured Postmark to perform 80,000 transactions on 40,000 files, forming a dataset of roughly 1-GB, spread over 10 subdirectories, with file sizes ranging from 4-KB to 28-KB, and read/write granularities of 4-KB. We report the transaction time, which excludes the initial file preallocation phase, for all cases.

FileBench is a flexible, application-level workload simulator. We used two predefined workload models to generate File Server and Web Server workloads. For the File Server workload, we configured FileBench to generate 10,000 files, using a mean directory width of 20 files, and a median file size of 128-KB. The workload generator performs a sequence of create, write, read, append (using a fixed I/O size of 1-MB), delete and stat operations, resulting in a write-biased workload. The Web Server configuration generates 25,000 files, using a mean directory width of 20 files. The median file size used is 32-KB, which results in a workload dominated by small file accesses, with the exception being an append-only log file. The workload generator performs a sequence of ten whole-file read operations, simulating reading web pages, followed by an append operation (with an I/O size of 16-KB) to a single log file. Unlike Postmark, the total data size generated by the FileBench workloads varies between 1.5-GB and 4-GB depending on the performance of the underlying caching system.

6.4.3 Comparative Evaluation: Caching Policies

Our first goal in evaluating the Loris prototype is to understand the impact of various caching policies on overall performance. Table 6.1 shows the IOPS/execution

time achieved by various benchmarks under Loris in the networked configuration. These results were obtained by fixing the host-side flash cache to 1.5-GB. There are three important observations to be made from Table 6.1.

First, notice that both write-back and write-through caching offer identical performance under Web Server. This is because of the fact that at 1.5-GB, the flash cache has a 100% read hit rate and all benchmark reads are satisfied entirely on the host side under both write-back and write-through schemes. The only writes under Web Server are appends to the log file which are never read back. Thus, this result proves that our caching algorithms work as expected.

Second, clearly, write-back caching has a significant impact on overall performance. It registers an impressive 310% improvement in IOPS under File Server and 66% reduction in execution time under Postmark. This clearly indicates the importance of using the host-side flash cache as a read-write cache (as opposed to a read-only cache).

Third, notice that write-around allocation policy consistently deteriorates performance of both write-back and write-through caching under all benchmarks. Without the write-around policy, all write misses allocate data in the SSD under both write-back and write-through caching policies. Thus, when the cache is large enough to hold a substantial portion of the working set, most read/write requests can be satisfied entirely on the host side. But with write around enabled, cache allocation happens only as a side effect of a read miss. Thus, the first read request for each data block has to be serviced by the networked file server, thereby reducing performance. As we found this to be the general case irrespective of the benchmark/experimental setting, we will not discuss write-around caching any further.

6.4.4 Network Performance Sensitivity

One of the rationales used by proponents of write-through caching is that the performance advantage of write-back caching would only be marginal, at best, when deployed in a setting with high-speed network interconnects and storage backends. In order to understand the sensitivity of write-caching policies with respect to network performance, we reran the experiments on the host machine using a direct-attached SATA HDD instead of the network file server as the storage backend. In this configuration, we used the default Loris physical module we described earlier to manage the layout of both SSD and HDD.

Looking at the results under cache size 1536-MB in Figures 6.6, 6.7, and 6.8, one can see that write-back caching still produces a 56% reduction in execution time under Postmark and a 234% increase in IOPS under the File Server benchmark. Based on these results, we believe that write-back caching is valuable even when used in a setting with low-latency interconnects and high-performance storage backends.

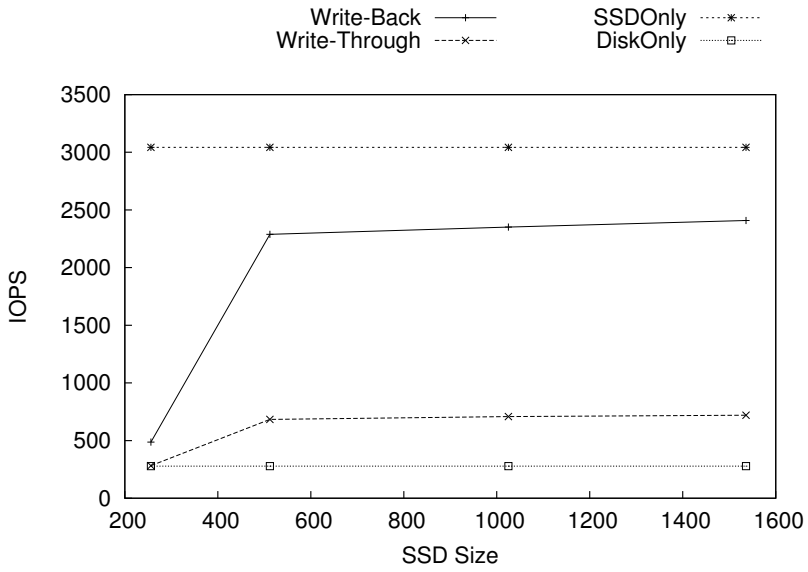


Figure 6.6: Figure shows the IOPS achieved by Loris under both single-device (HDD/SSD) and multi-device caching configurations at various cache sizes in MBs (axis labels) under the File Server benchmark.

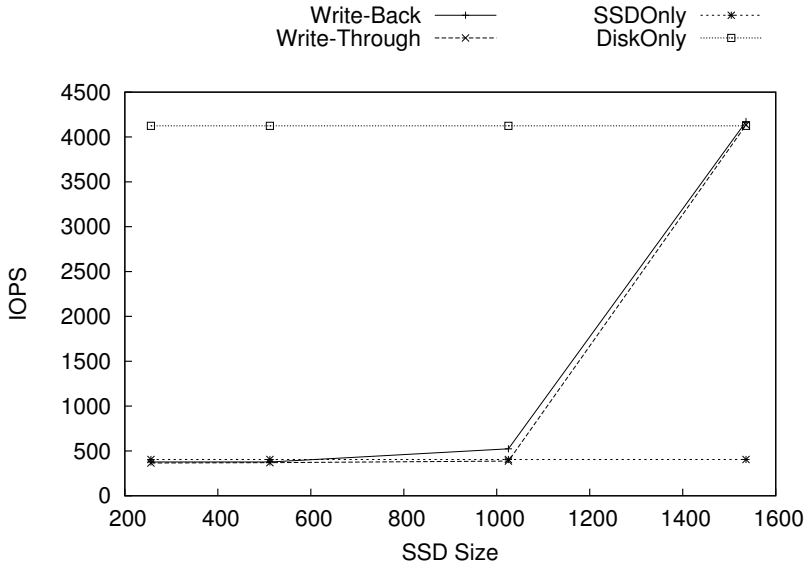


Figure 6.7: Figure shows the IOPS achieved by Loris under both single-device (HDD/SSD) and multi-device caching configurations at various cache sizes in MBs (axis labels) under the Web Server benchmark.

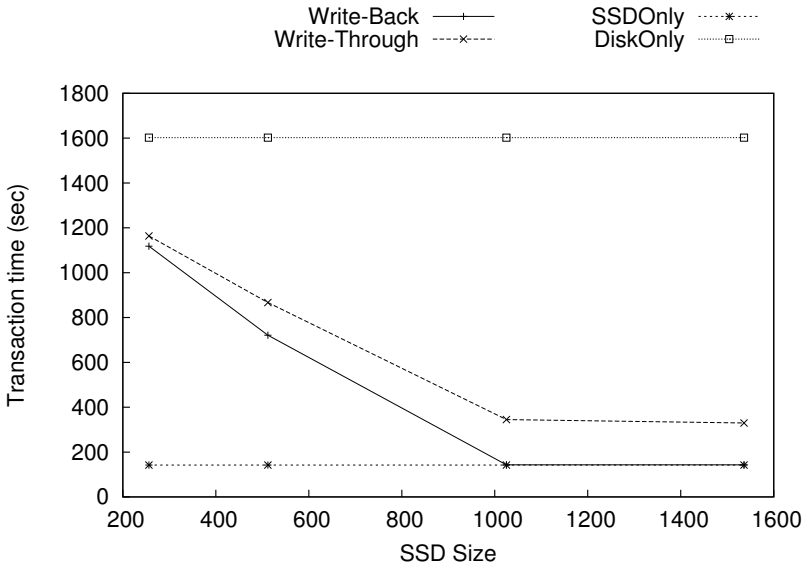


Figure 6.8: Figure shows the transaction time achieved by Loris under both single-device (HDD/SSD) and multi-device caching configurations at various cache sizes in MBs (axis labels) under the PostMark benchmark.

6.4.5 Cache Size Sensitivity

While write-back caching would have a clear edge over its write-through counterpart at cache sizes where the read hit rate is 100%, it is important to understand if it is still beneficial at smaller cache sizes. In order to do so, we reran the experiments in the direct-attached HDD configuration while varying the flash cache size. Figures 6.6, 6.7, and 6.8 report the IOPS/execution time of write-back/write-through caching policies under various benchmark-cache size combinations.

There are three important observations. First, one can see that even at the smallest cache size, write-back caching produces a 30% reduction in execution time under PostMark and a 75% increase in IOPS under File Server when compared to the disk-only case, thereby proving the utility of caching.

Second, as exemplified by the Web Server benchmark, read-intensive benchmarks derive no benefit from write-back caching. This is expected as the only writes under this benchmark are those issued to the append-only log that is never read. Thus, one might as well resort to using write-through caching, perhaps even with existing block-level solutions, under such read-intensive benchmarks.

Third, under write-intensive benchmarks, write-back caching always matches or outperforms write-though caching, but the performance of write-back caching is extremely sensitive to both the read/write ratio and cache size. For instance,

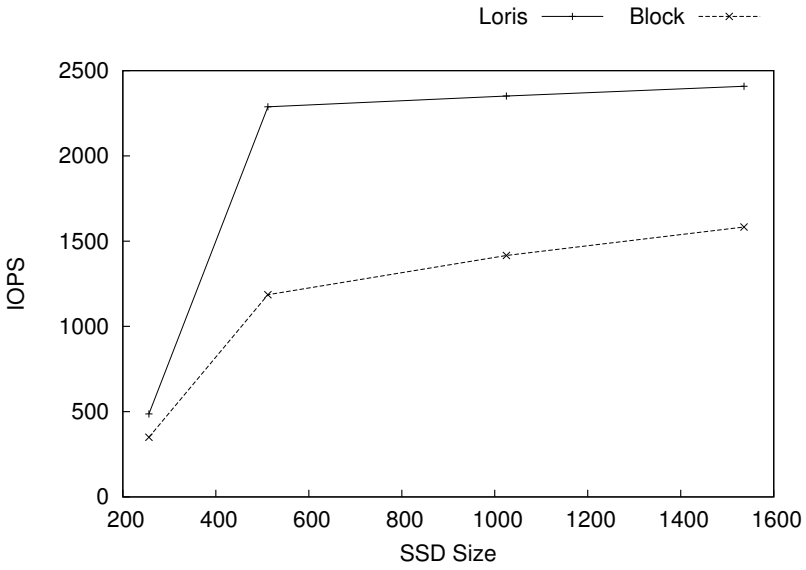


Figure 6.9: Figure shows the IOPS achieved by both Loris and block-level write-back caching implementations at various cache sizes in MBs (axis labels) under the File Server benchmark.

at cache size of 256-MB, while IOPS increases by a sizeable 72% under the File Server benchmark with write-back caching, Postmark results are more modest. However, at all other cache sizes, write-back caching improves execution time by 16% to 63% under Postmark.

6.4.6 File-Level and Block-Level Caching Comparison

One of the goals in evaluating our Loris prototype was to prove the effectiveness of file-level caching as opposed to block-level caching. In order to do so, we implemented a LRU-based caching filter driver that is positioned between the file system and AHCI disk driver. The cache driver works similar to its file-level counterpart by maintaining an in-memory list of disk blocks in LRU order. It implements write-back caching of data stored in the direct-attached HDD by interposing file system requests and satisfying both reads and writes from the SSD if possible. To make a fair comparison, we ran Loris over the block-level cache. Thus, in this configuration, Loris is not used as a caching framework, but rather as just a regular file system that manages the logical disk exposed by the block-level cache implementation.

Figures 6.9, 6.10, and 6.11 show the performance of the two caching implementations. Clearly, the Loris-based file-level caching implementation out-

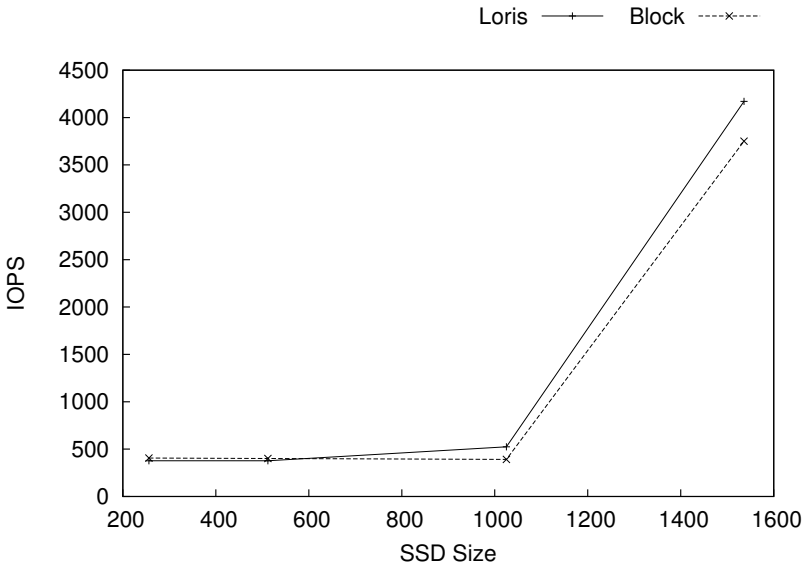


Figure 6.10: Figure shows the IOPS achieved by both Loris and block-level write-back caching implementations at various cache sizes in MBs (axis labels) under the Web Server benchmark.

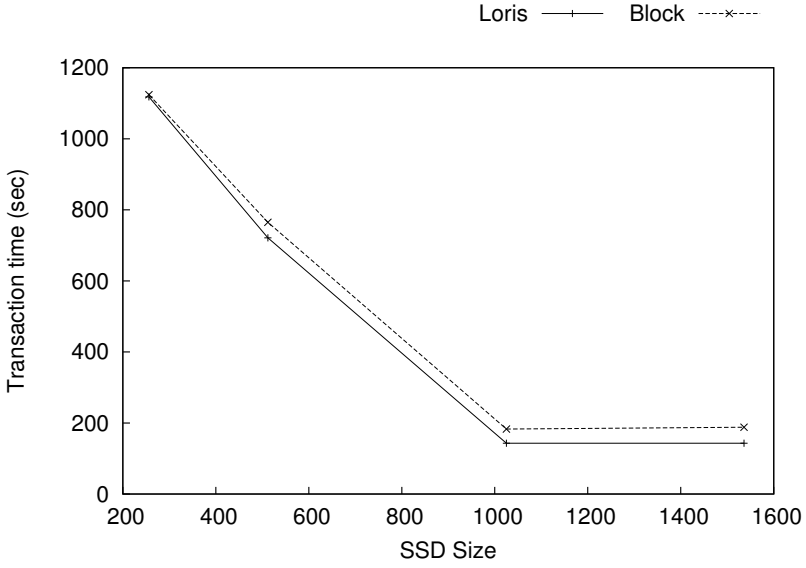


Figure 6.11: Figure shows the transaction time achieved by both Loris and block-level write-back caching implementations at various cache sizes in MBs (axis labels) under the PostMark benchmark.

performs its block-level counterpart by a significant margin under both Postmark at high cache sizes and under File Server at all cache sizes. On further investigation, we found out this to be due to the impact of file deletions on the two caching implementations. When a file is deleted, the Loris-based implementation frees all cached data associated with it in the SSD. In the block-level case, on the other hand, deletions are handled by the file system. Hence, the block-level cache driver never gets a notification from the file system about deleted file blocks. Thus, unlike Loris, which never migrates deleted data blocks, the block-level implementation incurs heavy penalty due to unwarranted caching and migration of useless deleted data. While this drawback could be overcome by having the file system share such semantic information with a block-level cache (using the TRIM command for instance), we would like to emphasize here that a file-level implementation has easy access to such rich semantic information out of the box. The conclusion we would like to draw from these results is that file-level caching implementations are capable of meeting the performance of their block-level counterparts without resorting to suboptimal, redundant consistency management.

6.5 Conclusion

As flash devices grow in density, we believe that write-back caching will become a standard feature in all future host-side caching systems. In this paper, we showed how a file-level integration of caching algorithms solves, by design, various consistency issues that plague the traditional block-level integration. Using our Loris prototype, we dispelled the myth that file-level caching implementations cannot work efficiently on a subfile basis. Although we used Loris as our framework, we would like to point out that one could theoretically build file-level caching systems using the traditional storage stack with the assistance of the stackable file system framework [43].

Given the very many benefits of file-level caching, an interesting area of future research is investigating its integration in modern virtualized data centers, where NAS-based filers are being increasingly deployed as backend stores for storing disk images of consolidated server virtual machines. Recent research has shown that unwarranted translations enforced by layers of virtualization have a huge negative impact on performance, and that such overhead could be eliminated by using a paravirtualized NAS client in the guest OS [98]. Thus, an alternative to the traditional approach of having the hypervisor perform caching of VM-disk-image blocks [18] would be to use a hypervisor-resident, file-level flash-caching implementation (like Loris) in combination with the paravirtualized NAS client to cache files rather than blocks. We intend to investigate the understand the pros and cons of such an approach as a part of future research.

Chapter 7

Discussion

As we mentioned in Chapter 1, this thesis is organized as a collection of refereed publications. In this section, we will elaborate on certain aspects of Loris that were not discussed in detail in these publications due to lack of space.

7.1 Metadata Management

Each Loris layer maintains and manages metadata relevant to its operation independent of other layers, as the requirements regarding persistence and reliability of metadata differ significantly across layers. For instance, the Cache layer uses metadata to track file pages currently buffered in memory. This metadata is dynamically updated when file pages are staged into (or evicted from) memory and is not persisted as the DRAM-based page cache is itself volatile. In contrast to this, the metadata managed by the Logical layer, that maps logical to physical files, for instance, needs to be stored reliably across reboots, power failures, and system crashes.

Such independent metadata management raises two questions that have not been answered elaborately in previous chapters: How is metadata consistency maintained in Loris? What are the reliability guarantees offered by metadata management? As we mentioned in Chapter 1, these research issues were deliberately left out of this thesis as they form a part of another thesis that focuses on reliability extensions to Loris. Having mentioned that, we will now discuss certain reliability aspects in order to clarify the interaction between features described in this thesis and reliability extensions.

7.1.1 Metadata Consistency

Typically, dirty metadata blocks buffered by each layer are flushed to stable storage on two occasions: 1) during cache eviction to make room for staging new

metadata blocks, 2) during a sync operation. As each layer dictates when layer-specific metadata is evicted and flushed to persistent storage, it is likely that the global state of on-disk metadata will be inconsistent at any given point in time. Hence, in the absence of any consistency mechanism, a system crash or a power failure would leave the on-disk metadata in an inconsistent state, and might cause data loss.

To protect metadata against such inconsistencies, Loris implements a coarse-grained consistency mechanism that uses system-wide metadata snapshoting to atomically move cross-stack metadata from one consistent on-disk state to another at well-defined synchronization points. In our current prototype, this synchronization point intentionally coincides with the periodic, five-second, POSIX sync call issued by the MINIX update daemon. On receiving the sync call, the Loris Naming layer finishes all ongoing operations, flushes all dirty metadata blocks by writing them out to the Cache layer, and finally, forwards the sync call down to the Cache layer. The Cache layer, in turn, flushes all dirty blocks and forwards the sync call to the Logical layer.

On receiving the sync call, the Logical layer performs three steps. First, it flushes all dirty metadata blocks. Second, it forwards the sync call to all Physical layers. After all the Physical layers complete the sync call successfully, the Logical layer initiates the third step by asking all Physical layers to snapshot the current on-disk state and tag the snapshot using a common timestamp. A successful completion of this last step moves the entire system to a new, consistent on-disk state. A failure, on the other hand, results in the Logical layer coordinating roll back by directing all Physical layers to switch to the last common timestamp. Thus, the sync-based consistency mechanism essentially wraps all system metadata in a single transaction which spans the period between two sync calls, and ensures that all metadata modifications in that transactions commit or rollback atomically.

One side effect of the sync-based checkpointing approach used by Loris is the lack of support in our current prototype for the POSIX fsync system call (MINIX 3 VFS does not support this call either). Supporting fsync is complicated, as selectively flushing data corresponding to a single file requires selective metadata flushing, which, in turn, requires explicitly tracking dependencies between various different types of metadata (directory blocks, inodes, bitmap blocks, etcetera) managed by different layers. We would like to point out here that the fsync issue is not just Loris specific as it is applies to other file systems as well. Rather than tracking and maintaining complicated dependency relationships, most file systems solve the fsync problem by a) using dedicated intent logs for logging application data, b) journaling data in addition to metadata, c) buffering data in non-volatile memory, or d) using fine-grained application-driven transactions. All these solutions are equally applicable to Loris and we leave the task of identifying the right division of labor between layers for implementing low-overhead fsync as future work.

7.1.2 Metadata Reliability

Layers in the Loris stack can be classified into two types depending on the mechanism used to protect layer-specific metadata, namely, stack-dependent layers, and self-managed layers. We will elaborate on these two types now. Recall that each Physical layer implementation in Loris is required to support some form of parental checksumming, which is used for verification of data on all reads. As all the layers above the Physical layer store layer-specific metadata in physical files, they can rely on the checksumming provided by the Physical layer to detect data corruption. Similar, the per-file RAID algorithms implemented in the Logical layer can be used by Loris to mirror all metadata across all available physical modules. Thus, metadata generated by all layers above the Logical layer, including Logical layer's own metadata, can be protected against device failures using RAID algorithms. Thus, Naming, Cache, and Logical layers are stack-dependent layers, as they Loris functionality typically used to protect application data is also used to protect system metadata for these layers.

In contrast to the three upper layers, Loris's Physical layer has to use custom logic to protect layer-specific metadata. As described in Chapter 3, the Physical layer can use the parental checksumming implementation to protect its own metadata against silent data corruption. However, in order to be able to recover metadata when corruption occurs, the Physical layer must explicitly manage redundancy of metadata blocks, as it cannot rely on file-level RAID algorithms implemented in the Logical layer to provide such redundancy. While one might perceive the duplication of redundancy logic in the logical and Physical layer as a drawback of the Loris layering, in reality, such is not the case the case due to three reasons.

First, compared to the Logical layer, which needs to implement sophisticated parity-based RAID algorithms that dynamically allocate and free data across several devices, the redundancy requirements of the Physical layer are substantially simple as they are specific to a single device. For instance, space for replicas of all metadata blocks can be allocated once when a device is first formatted.

Second, replication logic in the Physical layer must account for device-specific failure characteristics that need not be taken into account by the Logical layer RAID algorithms. For instance, two copies of metadata blocks must be stored in non-sequential disk blocks, preferably separated by a large distance in the LBA address space, to prevent correlated sector errors. Mirrored copies of a data block generated by RAID algorithms, in contrast, could be stored on any physical file in two different physical modules without any constraint on the exact on-disk location.

Third, the Physical layer needs to selectively replicate only a subset of its metadata, as a) all metadata belonging to higher levels are replicated on several devices, and b) logical redundancy can be used to recover several metadata blocks

without physical redundancy. For instance, it is not necessary to replicate directory data blocks within a physical module as they are mirrored across several devices. Similarly, if a bitmap block is identified to be corrupt on a read operation, the Physical layer could recover that block in the background by using block allocation information present in inodes, thus, obviating the need to maintain redundancy. Thus, given that the redundancy requirements for layout metadata are different enough from other stack metadata, we believe that adding customized logic does not violate our modularity principle.

Having said that, the current Physical layer implementation used in our prototype does not protect its own metadata using redundancy. Thus, a corruption in certain critical layout metadata, like the root inode block, would render an entire physical module unusable until the corruption is resolved. Note, however, that failures in other metadata blocks, like individual inode blocks, would be propagated as a checksum error to the Logical layer, which, in turn, would reissue the request to other physical modules and satisfy the application request. An interesting direction we are exploring as a part of future work involves extending the snapshoting mechanism in the Physical layer to provide sufficient redundancy in such situations.

7.2 Flash Management

We will now discuss certain aspects of flash-based storage tiering and caching that were omitted in Chapters 5 and 6 due to lack of space.

7.2.1 Tiering

In Chapter 5, we showed how Loris could be used as a framework for evaluating hybrid storage configurations that use flash-based SSDs in concert with HDDs in the context of an enterprise storage server. As our focus was on comparing the performance of various tiering and caching policies under a range of server workloads, we left open several aspects as future work. For instance, our Loris-based framework was only capable of performing whole-file caching or tiering due to a limitation of the Logical layer's mapping. This inability of Loris to map parts of a file to different storage targets made it unsuitable for certain workloads, like Virtual Desktop Infrastructure workloads, where different portions of a single large file (like Virtual Hard Disks) exhibit different access characteristics.

Since that work, we have extended Loris in several ways to implement other features. One such extension is the work presented in Chapter 6 which demonstrates the utility of Loris as a persistent, client-side cache in virtualized data centers. In order to support write-back caching, we modified Loris's Logical layer described in Chapter 5, which originally used a logical file as the granularity of

mapping, by switching to a new, fine-grained mapping that multiplexed logical files across physical files at the granularity of a 4-KB block. Although we used this new mapping only for evaluating write-back and write-through caching in a client context, it would be trivial to extend Loris and implement subfile tiering. In such a case, using the same subfile mapping presented in Chapter 6, Loris could easily map each logical file block, or an logical file extent, to a different tier depending on its usage characteristics. However, doing so also requires modifying the mechanism used for collecting and maintaining access statistics.

Originally, as described in Chapter 5, Loris used a separate in-memory array of access counters, one per file, in order to perform its data classification task. This approach had two benefits. First, by maintaining access statistics on a per file basis, we reduce the amount of metadata substantially compared to block-level systems that have to maintain per-block access counters. For instance, maintaining per-file statistics for an installation with a billion 1-GB files (1-EB of data) would require roughly 4-GB of additional memory (assuming a 4-byte access counter). A per-block (or per-extent) mechanism, in contrast, would need 4-TB even with an extent size of 1-MB. Second, by decoupling this metadata from the file mapping metadata, and by maintaining access statistics only in memory, we trade off the persistence of access statistics after a power failure for steady-state performance and ease of implementation. This trade off was made based on two observations. First, the loss of file classification information on power failure or system crash does not affect correctness, as file mapping metadata is always protected in such scenarios. Second, even though such a failure results in the loss of access history, data cached or tiered in the SSD continues to be serviced by the SSD, as file mapping information is preserved across reboots.

Despite its advantages, there are scenarios, like the VDI use case we mentioned earlier, where Loris would benefit from fine-grained subfile access tracking. However, tracking access statistics on a subfile granularity requires maintaining these statistics for each logical file block or file extent, which, as we mentioned earlier, poses a scalability issue when in-memory data structures are used. Thus, as a part of future work, we intend to extend Loris to support tracking and recording access statistics using other compact data structures that spill to secondary storage like external heaps.

7.2.2 Caching

The current Loris prototype supports the association of different write policies with system metadata and application data. An interesting aspect of Loris that was not described in detail in this thesis is the resilience offered by various write policies with respect to client-side SSD-cache failures. By default, Loris uses write-back caching policy for both metadata and user data. This configuration was chosen to derive a fair comparison between the Loris and the block-level caching

implementation described in Chapter 6, which, unlike Loris, did not even offer consistency guarantees in the face of system crash or power failure. Although this default Loris configuration offers protection against power failures, it has the obvious drawback that it fails to protect metadata and user data from client-side caching device failures.

As we mentioned Chapter 6, Loris can easily protect metadata against such failures by switching to a configuration which uses a write-through caching policy for all system metadata, thanks to the file-awareness of layers, and attribute-based out-of-band hinting infrastructure present in Loris. With this configuration, a loss of the caching device would only result in the application data loss, as all updates to metadata from higher layers are forwarded to both the cache device and the networked primary data store. It is important to note here that Loris never returns corrupt data to an application even though such failures might render metadata out of sync with application data. For instance, consider a size-increasing append to a file that is buffered by the SSD cache. While the actual data block is written only to the SSD, the metadata reflecting the new file size would be written through to the networked primary store. If the SSD fails before the dirty data block is synchronized to the primary store, the file size stored in the system metadata becomes invalid. However, when a read request for this file is processed by the Logical layer, it identifies a mismatch between the logical file size stored in the file mapping and the actual size of the physical file stored in the data store. As a result, any read request for the lost data block would result in Loris returning an error to the application. Similarly, overwrites to existing data that are lost due to SSD failures would also be identified by the Logical layer, as the file mapping information would identify these data blocks as dirty blocks that have not yet been synchronized.

While data loss in the face of device failures might be an acceptable trade off for some applications, certain others with non-zero Recovery Point Objective (RPO) might demand strict consistency even in the face of caching device failures. For these applications, Loris can be parameterized to selectively support write-through caching for application data in addition to system metadata. However, it is important to note that protecting from both power failures and device failures requires using write-through caching in combination with checkpointing of application data. Thus, depending on the type of mechanism used for checkpointing, it might not be feasible to implement such stringent failure resilience without sacrificing performance. This is certainly true with the current checkpointing mechanism supported by Loris [102] which is optimized for metadata-only checkpointing. In the future, we intend to implement a copy-on-write-based Physical layer in Loris similar to NetApp's Write Anywhere File Layout [45] and investigate the performance impact of using this "write-through-and-checkpoint-all" configuration.

Chapter 8

Summary and Conclusion

The storage hardware landscape has changed dramatically in the last five decades. New devices with interfaces and price/performance/reliability tradeoffs radically different from traditional block-based, magnetic storage media are now commonplace. Application demands on storage systems have also changed substantially. Modern storage systems are expected to be reliable in the face of crashes, power outages, hardware failures and even disasters, flexible in on-demand provisioning, allocation and removal of storage, and offer high performance under a variety of workloads.

The storage stack found in most operating systems today is the product of several decades evolution to accommodate these changes in hardware landscape and application requirements. Although the stack has grown horizontally, due to addition of several protocols within a layer, and vertically, due to the addition of new layers, its evolution has been driven by a single overarching factor – compatibility with existing installations. The result of such compatibility-driven evolution is a protocol layering that is riddled with outdated design assumptions that get invalidated time and again with the addition of each new functionality.

In this thesis, we outlined several performance, flexibility, reliability, and heterogeneity problems that plague the traditional stack and presented Loris, a new storage stack that solves all these problems by design. We will now summarize the key results of our research, following which, we will highlight potential areas where further research is required.

8.1 Summary

Fundamental to the design of Loris is the file-level integration of all storage protocols traditionally implemented at the block level, and the modular division of labor that enables storage protocols in each layer to evolve independently.

In Chapter 2, we presented the division of labor among layers in the Loris stack for implementing reliable RAID algorithms. We showed how the modular isolation of layout and RAID algorithms makes Loris heterogeneity friendly, as layout algorithms can be paired with devices while using a common RAID implementation. By assigning the role of data integrity verification to the lowest layer, we showed how end-to-end data integrity can achieved without redundant, cross-layer data verification. We highlighted the benefit of file-level RAID integration by showing how advanced reliability features like graceful degradation can be realized easily.

In Chapter 3, we presented our extensions to the Loris stack for implementing file volume virtualization and device management. We introduced File Pools, our new storage model, and demonstrated how it radically simplifies storage management. We showed how file-level integration of file volume management and storage management algorithms makes them device agnostic, thus, making them portable across several device families. We described the modular policy–mechanism split that enables the use of a single, efficient, fine-grained snapshoting implementation in the physical layer for realizing fine-grained per-file snapshoting and even open-close versioning. We also presented version directories, a unified interface for browsing through snapshoting history.

In Chapter 4, we demonstrated the benefit of Loris's modularity by showing how naming layer implementations can be changed, and can even co-exist, without modifying the rest of the storage stack. We showed how this modularity enables integration of domain-specific metadata management into the storage stack. With such an integration, we obviate the need for application-level metadata management, as the storage stack effectively functions as a high-performance database for storing, retrieving and querying metadata. Thus, Loris-based metadata management systems no longer suffer from performance or efficiency issues that limit the scalability of their application-level counterparts. Using Loris as a framework, we presented the design and implementation of a new naming layer that used customized LSM-trees for high-performance storage and an attribute-based query language for facilitating scalable metadata search.

In Chapter 5, we showed how Loris can be used as a framework for designing hybrid storage systems that use flash-based SSDs in concert with HDDs for providing high-performance, high-density primary data storage. We implemented several caching and tiering algorithms using the Loris as a framework and presented a comparative performance evaluation, highlighting pros and cons of each approach.

In Chapter 6, we considered the problem of integrating flash-based SSDs as second-level host-side caches in modern data centers. We showed how the block-level flash-caching solutions suffer from poor reliability due to lack of crash consistency. Using Loris as a framework, we made the case for file-level integration of flash caching algorithms and showed how consistency management techniques

that protect locally stored data from power outages or system crashes can be extended to cover cached data and networked primary data as well. As a natural extension to the work presented in Chapter 5, where we used Loris to implement several whole-file caching and tiering algorithms, we designed and implemented extensions to Loris's logical layer for performing efficient, subfile mapping of logical file data to different storage targets. Using these extensions, we implemented both write-back and write-through variants of second-level flash caching, and presented a comparative evaluation of Loris-based, file-level and traditional, block-level caching implementations to prove the efficiency of file-level caching.

We hope our work on Loris inspires storage system designers to consider file-level integration of storage protocols in designing next-generation storage systems. We believe that the compatibility benefit offered by the traditional block-level integration of storage protocols is not reason enough to ignore the performance, reliability, flexibility, and heterogeneity pitfalls. The retirement of the traditional storage stack is long overdue and this delay has resulted in the development of other parallel, custom-made stacks for dealing with specific device families. For instance, Linux has an entire storage stack dedicated for managing byte-granular flash devices to work around the limitations of the traditional storage stack [1]. We believe that a clean-slate approach to storage protocol layering is required now, more than ever, to prevent such redundant effort.

8.2 Future Work

We believe that our research on alternative protocol layering in the storage stack opens up several venues for future research.

8.2.1 Alternative Storage Stacks

Loris is certainly not the last word in storage stack design. The protocol layering used in Loris is only one of many other possible arrangements [6], [3]. As we mentioned earlier in this thesis, while these other stacks do employ alternative layering to solve some of the reliability or flexibility issues, Loris is the only stack to solve all issues by design while remaining heterogeneity friendly. Although we used Loris as the framework for implementing various file-level storage protocols, we believe that our research is not just Loris specific. An interesting avenue of future research involves investigating the integration of our file-based protocols in other storage stacks. We will give two such examples.

Stackable File Systems

File system stacking has long been touted as an alternative approach for adding new storage protocols to the traditional stack [43], [108]. In contrast to the tra-

ditional approach of adding protocols to the block-level RAID layer, file system stacking, as the name implies, involves layering several custom-tailored file systems, each implementing a storage protocol, between VFS, which forms the top-most layer, and an on-disk file system, which forms the bottom-most layer. Thus, theoretically speaking, all file-level storage protocols we designed and implemented in thesis can be ported to the traditional storage stack using the stackable file system framework. Pushing the idea to its extreme, one could even transform the traditional stack into the Loris stack using just file system stacking. However, realizing this practice necessitates overcoming certain performance and reliability issues like maintaining cache coherence and avoiding redundant data caching across file system layers, and investigating a unified approach for protecting all layers from crashes and power failures [107].

Object-Based Storage

Originally proposed as a solution for achieving direct-access to networked data (similar to a Storage Area Network protocol) without sacrificing NAS-like security [31], the object-based storage interface is being considered as a potential replacement for the semantically-impoverished block interface [77], [50]. Using this new interface to address flash-based SSDs is essentially a radical shift in the division of labor between file and storage systems, as device-specific layout management, originally performed by file systems, will now be performed by the device controller firmware. Thus, the integration of such devices into the traditional storage stack is clearly out of question.

Unlike the block-level integration of storage protocols in the traditional stack, Loris's file-level RAID, volume management, and caching protocols can be used unaltered with the new object-based storage device interface (OSD). Thus, the entirely software-based Loris stack we presented in this thesis can be extended to span across both OS and device firmware. Similarly, the file-level protocols we presented here can be recast and implemented in the context of an OSD controller that manages several direct-attached OSDs using object-based (rather than file-based) storage protocols [105].

8.2.2 Integrating New Storage Protocols

Although we used Loris to implement several features typically expected from modern storage systems, there are still several others which are yet to be integrated. For instance, several enterprise systems support encryption for providing secure storage and compression or deduplication for improving storage utilization. Integrating these new protocols into the Loris requires careful deliberation as they pose new design challenges.

Size Changing Algorithms

Unlike the protocols we implement in this thesis, compression is size changing in nature, as it does not perform a one to one mapping between a logical file block and a physical file block. Recent research has shown that such size-changing algorithms can be integrated, albeit with some performance penalty, at the file level using the stackable file system framework [109]. Further research is required to identify if a well-thought-out division of labor and policy–mechanism split between Loris layers can help ameliorate the performance penalty.

Deduplication

Although recent research has argued in favor of whole-file deduplication [65], there are certain use cases, like virtual desktop infrastructure (VDI) installations, where block-granular deduplication is certainly worth the effort [27]. Implementing scalable deduplication at the block level is a challenging endeavor in itself due to the size of the index that tracks the logical–physical block relationships. Integrating deduplication protocols at the file level increases the index size as we must now map each logical file block to the corresponding physical file block. Thus it is important to investigate the impact of this increased metadata footprint on performance and scalability, and if auxiliary datastructures, like bloom filters [110], can assist in speeding up index lookup even at the file level.

Although file-level integration of these storage protocols is challenging, it also creates several opportunities for optimizations hard to achieve in a block-level setting. Powered by semantic awareness, one could easily pair compression algorithms with file types, or use different encryption schemes based on data importance, or eliminate deduplication-induced defragmentation by grouping file blocks together.

8.2.3 Extensions to Existing Protocols

As our goal in this research was to demonstrate the benefits of integrating various protocols at the file level, we made certain assumptions, also made by other comparable, state-of-the-art academic research projects, about the target storage installation or application workloads to simplify implementation effort. Thus, an obvious area of future research involves implementing advanced versions of these protocols for supporting multitenant storage installations and multilevel host-side caches.

Integrating Flash-Based SSDs

As we already mentioned, virtualization-driven server consolidation is a norm in modern data centers. Extending our Loris-based file-level cache to support multi-

ple virtualized workloads requires investigating several design parameters. First, as we mentioned in chapter 6, one needs to analyze the pros and cons of using a hypervisor-integrated file-level cache in concert with a paravirtualized NAS client, as opposed to the traditional approach of caching virtual machine disk images. Second, cache partitioning algorithms must be implemented to dynamically divvy up the flash cache among competing workloads based on SLOs or other performance metrics [84], [36]. Third, the interaction between caching and file volume snapshoting must be reexamined to eliminate any inefficiencies caused by redundant data caching [18].

Integrating Phase Change Memory

Despite the advantages of flash-based solid state storage when compared to magnetic media, its eccentricities, like asymmetric read/write performance, limited lifetime, and density-scaling issues, have encouraged research on alternative memory technologies as potential flash, and possibly DRAM, replacement. Of these technologies, Phase Change Memory (PCM) has been touted as the "Unified Storage" medium capable of replacing both DRAM and flash due to its ability to offer persistent data storage with performance comparable to DRAM [57]. As the name indicates, PCM is a resistive memory technology that works by inducing a phase change in the storage material during writes which is detected during reads. Thus, unlike charge-based memory technologies like flash and DRAM, which require precise control over charge placement in floating gates or capacitors, PCM is expected to scale better due to the absence of charge-retention and leakage issues at low cell sizes.

Recently analyst predictions indicate that within the next few years, the cost/GB of PCM will drop to half that of DRAM, making PCM the second least expensive caching medium after flash [40]. Given such cost dynamics, and given the rapid adoption of virtualization-based server consolidation, servers equipped with gigabytes (or even terabytes) of PCM and flash are likely be used as clients in next-generation data centers for running storage-intensive workloads. As PCM has performance comparable to DRAM, researchers have shown that the optimal way of interfacing with it from a hardware point of view is to directly expose PCM-based storage to the CPU using the memory bus rather than hiding it behind the I/O controller and treating it as a storage device [20], [66].

An interesting area of future research is investigating the integration of PCM into the file-based Loris storage stack. A straightforward approach involves using PCM in place of RAM in the cache layer as a first-level data cache. While this approach would certainly simplify integration into the Loris stack, more research is required to understand if alternative layering brings any benefits. For instance, researchers have proposed hybrid file systems that combine PCM with flash in several capacities to improve performance and reliability [58], [106]. An alternative

approach for integrating PCM into the Loris stack would be designing one such physical layer that unifies the management of PCM and flash. Such a physical layer would obviate the cache layer reducing the Loris stack to just three layers. A third approach would be to directly expose PCM to applications by integrating PCM-management algorithms into domain-specific application libraries [103] that could also provide custom naming schemes. With such an integration, Loris would be reduced to just the bottom two layers used for managing the second-level flash cache.

References

[1] Memory technology devices. http://www.linux-mtd.infradead.org/.

[2] NetApp's Solid State Hierarchy. ESG White Paper, 2009.

[3] Avere Systems, The Avere Architecture for Tiered NAS, 2009.

[4] Compellent Harnessing SSD's Potential. ESG Storage Systems Brief, 2009.

[5] EMC Fast Cache - A Detailed Review. EMC White Paper, 2011.

[6] Sun Microsystems, Solaris ZFS file storage solution. Solaris 10 Data Sheets, 2004.

[7] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proc. of the USENIX Ann. Tech. Conf.*, 2008.

[8] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Block-level raid is dead. In *Proc. of the Second USENIX Work. on Hot topics in Storage and File Sys.*, 2010.

[9] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Loris - A Dependable, Modular File-Based Storage Stack. In *Proc. of the 16th IEEE Pacific Rim Intl. Symp. on Dep. Comp.*, 2010.

[10] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Flexible, Modular File Volume Virtualization in Loris. In *Proc. of 27th IEEE Conf. on Mass Storage Sys. and Tech.*, 2011.

[11] R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Integrating Flash-based SSDs into the Storage Stack. In *Proc. of 28th IEEE Conf. on Mass Storage Sys. and Tech.*, 2012.

[12] R. Appuswamy, D. C. van Moolenbroek, S. Santhanam, and A. S. Tanenbaum. File-level, host-side flash caching with loris. In *Proc. of the 19th IEEE Intl. Conf. on Parallel and Distr. Sys.*, 2013.

[13] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd: an optimal scheme for tolerating double disk failures in raid architectures. In *Proc. of the 21st Ann. Intl. Symp. on Comp. Arch.*, 1994.

[14] S. Bloehdorn, O. Görlitz, S. Schenk, and M. Völkel. TagFS – tag semantics for hierarchical file systems. In *Proc. of the Sixth Intl. Conf. on Know. Mgmt.*, 2006.

[15] S. Brandt, C. Maltzahn, N. Polyzotis, and W. Tan. Fusing data management services with file systems. In *Proc. of the 4th Ann. Work. on Petascale Data Storage*, 2009.

[16] A. Brown and D. A. Patterson. Towards availability benchmarks: a case study of software raid systems. In *Proc. of the USENIX Ann. Tech. Conf.*, 2000.

[17] A. B. Brown and D. A. Patterson. To err is human. In *Proc. of the First Work. on Eval. and Arch. Sys. Dep.*, 2001.

[18] S. Byan, J. Lentini, A. Madan, and L. Pabon. Mercury: Host-side flash caching for the data center. In *Proc. of 28th IEEE Conf. on Mass Storage Sys. and Tech.*, 2012.

[19] F. Chen, D. A. Koufaty, and X. Zhang. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proc. of the Intl. Conf. on Supercomp.*, 2011.

[20] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proc. of the 22nd ACM Symp. on Oper. Sys. Prin.*, 2009.

[21] J. Corbett. A bcache update. http://lwn.net/Articles/497024/, 2012.

[22] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proc. of the Third USENIX Conf. on File and Storage Tech.*, 2004.

[23] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *Proc. of the Fall Joint Comp. Conf.*, 1965.

[24] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the Information Gap in Storage Protocol Stacks. In *Proc. of the USENIX Ann. Tech. Conf.*, 2002.

[25] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Journal-guided resynchronization for software raid. In *Proc. of the Fourth USENIX Conf. on File and Storage Tech.*, 2005.

[26] J. K. Edwards, D. Ellard, C. Everhart, R. Fair, E. Hamilton, A. Kahn, A. Kanevsky, J. Lentini, A. Prakash, K. A. Smith, and E. Zayas. Flexvol: Flexible, efficient file volume virtualization in wafl. In *Proc. of the USENIX Ann. Tech. Conf.*, 2008.

[27] J. Feng and J. Schindler. A deduplication study for host-side caches in virtualized data center environments. *Proc. of the 29th IEEE Conf. on Mass Storage Syst. and Tech.*, 2013.

[28] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Comp. Surv.*, 37(2):138–163, 2005.

[29] G. R. Ganger, M. K. McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. Comp. Sys.*, 18(2):127–153, 2000.

[30] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of the 19th ACM Symp. on Oper. Sys. Prin.*, 2003.

[31] G. A. Gibson and R. Van Meter. Network attached storage architecture. *Commun. ACM*, 43(11):37–45, 2000.

[32] D. Gifford, P. Jouvelot, M. Sheldon, and J. O'Toole, Jr. Semantic file systems. In *Proc. of the 13th ACM Symp. on Oper. Sys. Prin.*, 1991.

[33] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The cedar file system. *Commun. ACM*, 31:288–298, 1988.

[34] C. Giuffrida and A. S. Tanenbaum. Cooperative update: a new model for dependable live update. In *Proc. of the Second Intl. Work. on Hot Topics in Software Upgrades*, 2009.

[35] B. Gopal and U. Manber. Integrating content-based access mechanisms with fierarchical file systems. In *Proc. of the Third USENIX Symp. on Oper. Syst. Design and Impl.*, 1999.

[36] P. Goyal, D. Jadav, D. Modha, and R. Tewari. Cachecow: Qos for storage system caches. In *Proc. of the Intl. Work. on Quality of Service*, 2003.

[37] J. Guerra, H. Pucha, J. Glider, W. Belluomini, and R. Rangaswami. Cost Effective Storage using Extent Based Dynamic Tiering. In *Proc. of the Ninth USENIX Conf. on File and Storage Tech.*, 2011.

[38] A. Gulati, M. Naik, and R. Tewari. Nache: Design and Implementation of a Caching Proxy for NFSv4. In *Proc. of the Fifth USENIX Conf. on File and Storage Tech.*, 2007.

[39] R. Hagmann. Reimplementing the cedar file system using logging and group commit. In *Proc. of the 11th ACM Symp. on Oper. Sys. Prin.*, 1987.

[40] J. Handy. Phase-change memory becomes a reality. http://objective-analysis.com/, 2009.

[41] L. G. Harbaugh. Storage Smackdown: Hard drives vs SSDs. http://www.networkworld.com/reviews/2010/041910-ssd-hard-drives-test.html, 2010.

[42] J. Harler and F. Oh. Dynamic Storage Tiering: The Integration of Block, File and Content, 2010.

[43] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Trans. Comp. Sys.*, 12(1):58–89, 1994.

[44] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a Highly Dependable Operating System. In *Proc. of the Sixth European Dep. Comput. Conf.*, 2006.

[45] D. Hitz, J. Lau, and M. Malcolm. File system design for an nfs file server appliance. In *Proc. of the USENIX Winter Tech. Conf.*, 1994.

[46] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comp. Sys.*, 6(1):51–81, 1988.

[47] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Proc. of the USENIX Ann. Tech. Conf.*, 2013.

[48] W. K. Josephson, L. A. Bongo, D. Flynn, and K. Li. Dfs: A file system for virtualized flash storage. In *Proc. of the Eigth USENIX Conf. on File and Storage Tech.*, 2010.

[49] N. Joukov, A. M. Krishnakumar, C. Patti, A. Rai, S. Satnur, A. Traeger, and E. Zadok. RAIF: Redundant Array of Independent Filesystems. In *Proc. of 24th IEEE Conf. on Mass Storage Sys. and Tech.*, 2007.

[50] Y. Kang, J. Yang, and E. L. Miller. Object-based scm: An efficient interface for storage class memories. In *Proc. of the 27th Symp. on Mass Storage Syst. and Tech.*, 2011.

[51] G. Karche, M. Mamidi, and P. Massiglia. Using Dynamic Storage Tiering. Tech. report, 2006.

[52] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Trans. Comp. Sys.*, 10(1):3–25, 1992.

[53] Y. Klonatos, T. Makatos, M. Marazakis, M. D. Flouris, and A. Bilas. Azor: Using Two-Level Block Selection to Improve SSD-Based I/O Caches. In *Proc. of the Sixth Intl. Conf. on Net., Arch., and Storage*, 2011.

[54] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, and M. Zhao. Write policies for host-side flash caches. In *Proc. of the 11th USENIX Conf. on File and Storage Tech.*, 2008.

[55] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dussea. Parity lost and parity regained. In *Proc. of the Sixth USENIX Conf. on File and Storage Tech.*, 2008.

[56] B. Laliberte. Automate and Optimize a Tiered Storage Environment - FAST! Tech. report, 2009.

[57] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *Proc. of the 36th Ann. Intl. Symp. on Comp. Arch.*, 2009.

[58] E. Lee, H. Bahn, and S. H. Noh. Unioning of the buffer cache and journaling layers with non-volatile memory. In *Proc. of the 11th USENIX Conf. on File and Storage Tech.*, 2013.

[59] A. Leung, I. Adams, and E. Miller. Magellan: A searchable metadata architecture for large-scale file systems. Tech. Report UCSC-SSRC-09-07, University of California, Santa Cruz, 2009.

[60] A. Leung, M. Shao, T. Bisson, S. Pasupathy, and E. Miller. Spyglass: fast, scalable metadata search for large-scale storage systems. In *Proc. of the Seventh USENIX Conf. on File and Storage Tech.*, 2009.

[61] A. W. Leung, S. Pasupathy, G. Goodson, and E. L. Miller. Measurement and Analysis of Large-Scale Network File System Workloads. In *Proc. of the USENIX Ann. Tech. Conf.*, 2008.

[62] A. Leventhal. Flash Storage Memory. *Commun. ACM*, 51(7), 2008.

[63] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[64] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Proc. of the Winter USENIX Conf.*, 1991.

[65] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. of the Ninth USENIX Conf. on File and Storage Tech.*, 2011.

[66] J. C. Mogul, E. Argollo, M. Shah, and P. Faraboschi. Operating system support for nvm+dram hybrid main memory. In *Proc. of the 12th USENIX Work. on Hot Topics in Oper. Sys.*, 2009.

[67] K.-K. Muniswamy-Reddy and D. A. Holland. Causality-based versioning. In *Proc. of the Seventh USENIX Conf. on File and Storage Tech.*, 2009.

[68] K.-K. Muniswamy-Reddy, C. P. Wright, A. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *Proc. of the Third USENIX Conf. on File and Storage Tech.*, 2004.

[69] M. Olson. The design and implementation of the Inversion file system. In *Proc. of the Winter USENIX Tech. Conf.*, 1993.

[70] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4):351–385, 1996.

[71] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the unix 4.2 bsd file system. In *Proc. of the Tenth ACM Symp. on Oper. Sys. Prin.*, 1985.

[72] S. Patil and G. Gibson. Scale and concurrency of giga+: file system directories with millions of files. In *Proc. of the Ninth USENIX Conf. on File and Storage Tech.*, FAST'11, 2011.

[73] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of the 1988 ACM SIGMOD Intl. Conf. on Management of data*, 1988.

[74] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *ACM Trans. on Storage*, 1:190–212, 2005.

[75] M. Polte, J. Simsa, and G. Gibson. Enabling enterprise solid state disks performance. In *Proc. of the First Work. on Integ. Solid-state Memory into the Storage Hierarchy*, 2009.

[76] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proc. of the 20th ACM Symp. on Oper. Sys. Prin.*, 2005.

[77] A. Rajimwale, V. Prabhakaran, and J. D. Davis. Block management in solid-state devices. In *Proc. of the USENIX Ann. Tech. Conf.*, 2009.

[78] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.

[79] D. Roselli, J. R. Lorch, and T. E. Anderson. A comparison of file system workloads. In *Proc. of the USENIX Ann. Tech Conf.*, 2000.

[80] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, 1992.

[81] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir. Deciding when to forget in the elephant file system. In *Proc. of the 17th ACM Symp. on Oper. Sys. Prin.*, 1999.

[82] M. Saxena, M. M. Swift, and Y. Zhang. Flashtier: a lightweight, consistent and durable storage cache. In *Proc. of the 17th ACM European Conf. on Comp. Sys.*, 2012.

[83] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *Proc. of the 12th Intl. Conf. on Dist. Comp. Sys.*

[84] P. Sehgal, K. Voruganti, and R. Sundaram. Slo-aware hybrid store. In *Proc of the 28th IEEE Symp. on Mass Storage Sysems and Tech.*, 2012.

[85] B. Sidebotham. Volumes: the andrew file system data structuring primitive. Tech. Report CMU-ITC-053, 1986.

[86] G. Sivathanu, C. P. Wright, and E. Zadok. Ensuring data integrity in storage: techniques and applications. In *Proc. of the ACM Work. on Storage Sec. and Surviv.*, 2005.

[87] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or death at block-level. In *Proc. of the Sixth Conf. on Symp. on Oper. Sys. Design & Impl.*, 2004.

[88] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with d-graid. *ACM Trans. on Storage*, 1(2):133–170, 2005.

[89] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber. Extending SSD lifetimes with disk-based write caches. In *Proc. of the Eighth USENIX Conf. on File and Storage Tech.*, 2010.

[90] C. A. Stein, J. H. Howard, and M. I. Seltzer. Unifying file system protection. In *Proc. of the USENIX Ann. Tech. Conf.*, 2001.

[91] L. Stein. Stupid file systems are better. In *Proc. of the Tenth USENIX Work. on Hot Topics in Oper. Systems*, 2005.

[92] J. Stender, B. Kolbeck, M. Hoĺgqvist, and F. Hupfeld. Babudb: fast and efficient file system metadata storage. In *Proc. of the Sixth Intl. Work. on Storage Net. Arch. and Parallel I/Os*, 2010.

[93] D. Stodolsky, G. Gibson, and M. Holland. Parity logging overcoming the small write problem in redundant disk arrays. In *Proc. of the 20th Ann. Intl. Symp. on Comp. Arch.*, 1993.

[94] StorageReview. Intel x25v ssd review. http://www.storagereview.com/intel_x25v_ssd_review_40gb, 2010.

[95] H. Tada, O. Honda, and M. Higuchi. A File Naming Scheme using Hierarchical-Keywords. In *Proc. of the 26th Ann. Intl. Comp. Soft. and App. Conf.*

[96] Taneja Group Technology Analysts. The State of the Core ? Engineering the Enterprise Storage Infrastructure with the IBM DS8000. White Paper, 2009.

[97] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems Design and Implementation (Third Edition)*. Prentice Hall, 2006. ISBN 0131429388.

[98] V. Tarasov, D. Hildebrand, G. Kuenning, and E. Zadok. Virtual machine workloads: The case for new benchmarks for NAS. In *Proc. of the 11th USENIX Conf. on File and Storage Tech.*, 2013.

[99] D. Teigland and H. Mauelshagen. Volume managers in linux. In *Proc of the USENIX Ann. Tech. Conf.*, 2001.

[100] M. Uysal, A. Merchant, and G. A. Alvarez. Using MEMS-Based Storage in Disk Arrays. In *Proc. of the Second USENIX Conf. on File and Storage Tech.*, 2003.

[101] R. van Heuven van Staereling, R. Appuswamy, D. C. van Moolenbroek, and A. S. Tanenbaum. Efficient, modular metadata management with loris. In *Proc. of the Sixth Intl. Conf. on Net., Arch., and Storage*, 2011.

[102] D. van Moolenbroek, R. Appuswamy, and A. Tanenbaum. Integrated system and process crash recovery in the loris storage stack. In *IEEE Seventh Intl. Conf. on Net., Arch. and Storage*, 2012.

[103] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th Intl. Conf. on Arch. Supp. for Prog. Lang. and Oper. Sys.*, 2011.

[104] A.-I. Wang, P. L. Reiher, G. J. Popek, and G. H. Kuenning. Conquest: Better Performance Through a Disk/Persistent-RAM Hybrid File System. In *Proc. of the USENIX Ann. Tech. Conf.*, 2002.

[105] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *Proc. of the Sixth USENIX Conf. on File and Storage Tech.*, 2008.

[106] X. Wu and A. Reddy. Scmfs: A file system for storage class memory. In *Intl. Conf. on High Perf. Comp., Net., Storage and Analysis*, 2011.

[107] E. Zadok. The layers are coming, the layers are coming. In *Proc. of the Linux Storage and File Syst. Workshop*, 2008.

[108] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *Proc. of the USENIX Ann. Tech. Conf.*, 1999.

[109] E. Zadok, J. M. Andersen, I. Badulescu, and J. Nieh. Fast indexing: Support for size-changing algorithms in stackable file systems. In *Proc. of the USENIX Ann. Tech. Conf.*, 2001.

[110] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of the Sixth USENIX Conf. on File and Storage Tech.*, 2008.

Samenvatting

Alle moderne besturingssystemen gebruiken een verzameling protocollen die uit meerdere lagen zijn opgebouwd om de bekende hiërarchische abstractie van het bestandssysteem aan applicaties te leveren. In overeenstemming met de literatuur zullen we hier verder naar verwijzen als de "opslagstack". Het opbouwen van protocollen uit lagen is niet specifiek voor de opslagstack. Zo gebruiken besturingssystemen bijvoorbeeld ook een gelaagde verzameling protocollen om netwerkfaciliteiten aan applicaties aan te bieden, ook wel aangeduid als de "netwerkstack". Het gebruik van lagen met vastgelegde abstracties er tussenin heeft in beide gevallen de onafhankelijke evolutie van protocollen mogelijk gemaakt, waarbij wijzigingen in één laag geen gevolgen hebben voor de andere lagen.

Er is echter een belangrijk verschil tussen de twee stacks: in tegenstelling tot de netwerkstack, waar de integratie van de protocollen gestandaardiseerd is op basis van vaste ontwerpen, is de integratie van de lagen van de opslagstack enkel gedreven door ondersteuning van oudere systemen.

In deze dissertatie bekijken we de traditionele opslagstack op basis van drie dimensies: betrouwbaarheid, flexibiliteit en heterogeniteit. We identificeren enkele problemen met de traditionele opslagstack in elke dimensie en laten zien hoe oude ontwerpbeslissingen zijn genomen op basis van aannames die door het toevoegen van protocollen aan de opslagstack voor compatibiliteit niet meer geldig zijn en hoe dit de oorzaak is van al deze problemen. Hiermee beargumenteren we waarom de traditionele opslagstack beter uit gebruik genomen zou kunnen worden, aangezien deze niet alleen ineffectief is in het beheer van hedendaagse opslagsystemen, maar ook niet voorbereid is om met de verwachte veranderingen in opslagoplossingen van de toekomst overweg te kunnen. Vervolgens presenteren we Loris, een herontwerp van de opslagstack op een schone lei die zodanig is ontworpen dat alle problemen van de traditionele opslagstack ermee opgelost worden.

In de opbouw van de lagen van het Loris protocol maken we de bewuste keuze om compatibiliteit met oude systemen op te geven zodat we de modulariteit en ondersteuning voor heterogene apparaten kunnen verbeteren. Loris verbetert de modulariteit door de monolitische bestandssysteemlaag uit de traditionele op-

slagstack op te delen in drie lagen, namelijk Naamgeving, Cache en Indeling. Daardoor kunnen protocollen in één laag vervangen worden zonder dat de andere lagen hierdoor beïnvloed worden. Wij hebben van deze mogelijkheid gebruik gemaakt om domein-specifieke naamgevingsprotocollen te implementeren in Loris.

Loris ondersteunt heterogene hardware door nadrukkelijk apparaat-agnostische opslagprotocollen op het bestandsniveau te implementeren in plaats van apparaat-specifieke protocollen die afhankelijk van het onderliggende opslagapparaat op het niveau van blokken, bytes of objecten geïmplementeerd zijn. Het resultaat van de nadruk op modulariteit en heterogeniteit in het ontwerp is een opslagstack waarin alle lagen kennis hebben van bestanden en alle protocollen werken op basis van bestanden in plaats van blokken.

Deze dissertatie omschrijft een mogelijke realisatie van Loris: een op software gebaseerde protocol stack die draait op het MINIX3 microkernel besturingssysteem. Hiermee leveren we de volgende bijdragen:

1. Nieuwe opslagstack

    - Het ontwerp, de implementatie en de evaluatie van de lagen van het Loris protocol samen met een diepgaande vergelijking van de Loris protocollen met hun equivalenten in de traditionale opslagstack.

2. RAID op bestandsniveau en volume management

    - File Pools, een nieuw opslagmodel dat het beheer van apparaten eenvoudiger maakt
    - Een op Loris gebaseerde virtualizatie-infrastructuur die ondersteuning biedt om thin-provisioned bestandsvolumes een enkele file pool te kunnen laten delen
    - Een enkele geïntegreerde op Loris gebaseerde infrastructuur voor het maken van momentopnames van volumes en bestanden en het bijhouden van meerdere versies per bestand met een duidelijke scheiding tussen beleid en mechanisme

3. Efficiënt beheer van metadata

    - Een op Loris gebaseerd modulair systeem voor beheer van metadata dat 1) metadata opslag met behulp van LSM-trees, 2) infrastructuur om real-time te indexeren met gebruik van LSM-trees voor het bijhouden van indices van attributen, en 3) schaalbaar metadata zoeken met behulp van een op attributen gebaseerde zoektaal biedt

4. Flash-integratie op bestandsniveau

- Een op Loris gebaseerd raamwerk dat het mogelijk maakt om op bestandsniveau hybride primaire opslagsystemen te ontwikkelen die SSDs samen met HDDs gebruiken in verschillende rollen en een empirische studie van de voors en tegens van verschillende algoritmes voor caching en gelaagde opslag.
- Het ontwerp en de implementatie van een op Loris gebaseerde flash cache aan de kant van de host die fijnmazige caching uitvoert van delen van bestanden zonder problemen met de consistentie