End-to-End Abstractions for Application-Aware Storage

A Dissertation Presented

by

Gopalan Sivathanu

to

The Graduate School

in Partial fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-08-01

May 2008

Stony Brook University

The Graduate School

Gopalan Sivathanu

We, the dissertation committee for the above candidate

for the degree of Doctor of Philosophy, hereby recommend

acceptance of this dissertation.

Dr. Erez Zadok, Advisor
Associate Professor, Computer Science Department

Dr. R. Sekar, Chair-person of Defense
Computer Science Department

Dr. Rob Johnson
Assistant Professor, Computer Science Department

Dr. Remzi H. Arpaci-Dusseau
Associate Professor, Department of Computer Sciences,
University of Wisconsin-Madison

This dissertation is accepted by the Graduate School

Lawrence Martin
Dean of the Graduate School

Abstract of the Dissertation

End-to-End Abstractions for Application-Aware Storage
by

Gopalan Sivathanu

Doctor of Philosophy
in
Computer Science

Stony Brook University

2008

Modern computer systems are a composition of several logically independent layers. From a very simple hardware-software layering in early days, this layering in computer systems has increased significantly over time in both its depth and complexity. For example, in the past, file systems communicated directly with the disk hardware, whereas today layers such as logical volume managers, RAID, or even a network can exist in between. Although providing many important benefits, this rampant layering has also led to the well-explored problem of information-divide in the systems stack. Layers hide information, thus constraining functionality and limiting the power of individual layers. A particularly striking instance of this general problem exists in the storage stack today. Modern high-end storage systems have significant processing capabilities, but despite their potential, storage systems are constrained in their functionality because they are oblivious of knowledge about higher layers such as the applications using them.

In this thesis proposal, we seek to answer a simple question: *how can we convey application-level information across the diverse modern storage stack in a simple and generic manner?* We propose two flexible abstractions to solve this problem. The first abstraction we present is the notion of type-awareness in the storage stack. In type-aware storage, lower layers of the storage stack such as the disk are aware of the pointer relationships between disk blocks that are imposed by higher layers such as the file system. Type-awareness enables semantics-aware optimizations in the lower layers of the storage stack, and also active enforcement of invariants on data access based on the pointer relationships, resulting in better security and integrity. The second abstraction we evolve is Context-Aware I/O (CAIO), a generic mechanism to propagate information end-to-end through the storage stack. CAIO provides a simple, yet effective interface to communicate *application-data* and *application-I/O* relationships to the storage stack, enabling interesting functionality.

Through several case studies, we demonstrate the flexibility and benefits of both abstractions and show that they present a simple yet effective general interface to build the next generation of storage systems.

To my father, mother, and brothers.

Contents

List of Figures

List of Tables

Acknowledgments

I am first thankful to my advisor Erez Zadok for his great support and guidance throughout my five year period of PhD at Stony Brook. Erez played a great role in shaping up my research skills by giving me a lot of guidance on how to approach a new research problem, systematic performance evaluation, and formal paper writing. Erez provided me with the right set of opportunities such as attending several top conferences, internship referrals, and an excellent infrastructure in his lab. I am grateful to Erez for his trust and belief over my skills—the freedom of work I enjoyed during my PhD is remarkable. Erez's dedication towards the success of his students coupled with his strong research and teaching expertise make him a great advisor.

I thank R. Sekar, Remzi Arpaci-Dusseau, and Rob Johnson for serving in my defense committee. I am grateful to Sekar for his valuable feedback for several of my projects during my PhD. I am indebted to Remzi for his extraordinary help—he took the trouble of traveling to Stony Brook for my prelims. I have also been bugging Remzi asking for feedback for several of my ideas, and he was kind and patient enough to allocate time for it. Such helping tendency is remarkable.

My brother Muthian Sivathanu is my main inspiration to do a PhD. He is my role model and I have always strived to emulate him in his academic and research achievements. I am thankful to my brother for his valuable feedback in almost all of my projects, during the various stages of my PhD.

The comments and suggestions of the anonymous reviewers for my various submissions helped improve the quality of the thesis substantially. I thank Garth Gibson who shepherded my OSDI paper, for his extensive and detailed comments that helped improve the quality of content in this thesis.

It was a great experience collaborating actively with Swaminathan Sundararaman, Kiron Vijayasankar, and Chaitanya Yalamanchili, in several projects during my PhD. My interactions with Charles Wright during the intial stages of my PhD was quite helpful in getting accustomed to the culture and projects in the File Systems and Storage Lab. I am thankful to my colleagues, Sean Callanan, Abhishek Rai, and Avishay Traeger who joined in the same semester as I did, for their company and support. Other present and past members of the lab, Akshat Aranya, Jay Dave, Ramya Edara, Naveen Gupta, Puja Gupta, Rakesh Iyer, Nikolai Joukov, Aditya Kashyap, Harikesavan Pathangi Krishnan, Arun Krishnakumar, Devaki Kulkarni, Adam Martin, Kiran-Kumar Muniswamy-Reddy, Harry Papaxenopoulos, Chaitanya Patti, David Quigley, Sunil Satnur, Josef Sipek, Rick Spillane, Gopala Suryanarayana, Kumar Thangavelu, and Mohammad Nayyer Zubair, have helped reviewing my papers, or involved in brainstorming sessions.

I thank my mentors, managers, and colleagues during my three internships, for their contribution in making my summer internships highly enjoyable and educative. I particularly thank Omer Zaki, Kaladhar Voruganti, and KK Rao in IBM Almaden, Irfan Ahmad, Vikram Makhija, and Jennifer Anderson in VMware, and Gautham Thambidorai, Srdjan Petrovic, and Qixiang Sun in Google.

I am thankful to Tzi-cker Chieuh, Christos Karamanolis, Patrick McDanel, Ethan Miller, and Radu Sion for their feedback about my research.

I am fortunate to have gotten great friends at Stony Brook during my PhD. Specifically, I would like to thank Arvind Ramanathan, Prasanna-Kumar Thoguluva-Santharam, Vasudevan Srinivasan, Sumant Sankaran, Prithviraj Subburaj, and Sridhar Madishetty for making my experience at Stony Brook a memorable one.

Above all, mere words cannot express the boundless love, affection, support, and guidance extended to me by my father and mother all these years. All credits of my accomplishments goes to my parents for their unparalleled efforts in guiding me in the right path. I am blessed to have gotten brothers like Muthian and Sankaran—life without their love, support, company, and comforting is unimaginable.

This work was partially made possible by NSF CAREER EIA-0133589 and NSF CCR-0310493 awards.

Chapter 1

Introduction

Computer system design over the past years has revolved around the principle of layering [19]. Building systems as a hierarchy of layers enables localized and independent innovation in the individual layers. For example, in the network protocol stack comprising layers such as application, transport, network, and data link, each layer can independently innovate as long as the interface exported to the other layers is intact. With the growing complexity of today's systems, layering has become indispensable in hardware and software design.

Despite its obvious benefits, layered system design also comes with an inevitable side-effect: information available at one layer is not visible at the other layers beyond what is permitted by the interface separating those layers. The impact of this lack of information is becoming more pronounced in the recent years as there is a need for individual layers to support advanced functionality, requiring cross-layer information. This problem is exacerbated by the fact that recent advancements in computer systems such as virtual machine technology [7] have introduced more layers of virtualization in the systems stack, further widening the information-gap. Past research has identified this general problem in the context of different domains [4, 23, 55].

In the modern storage hierarchy, the general problem of information-gap between layers has hampered development of new functionality. Large-scale storage systems today comprise diverse resources that include high processing power, hundreds of gigabytes of RAM, solid state storage media such as flash, and hundreds or even thousands of disks [16, 37]. Despite this advancement in storage hardware, storage systems are constrained in the range of functionality they can provide, because they lack information about higher-level data semantics. Several solutions have been proposed to workaround this problem [5, 18, 53, 70, 71], but they are mostly limited in scope, and none of them have been widely adopted.

In this thesis, we evolve a new class of abstractions and mechanisms to effectively bridge the information-gap in the storage stack with minimal modifications to existing hardware and software infrastructures. Our solution is to propagate minimal *generic* information about application-level semantics across the storage stack, in an end-to-end manner. By decoupling the *generation* of such information from how the information is *used* in the storage stack, our solution provides a simple and general interface for better application-awareness in the lower layers of the storage stack. We specifically introduce two new abstractions under this category, to encode higher-level semantics: *type-awareness* and *context-awareness*.

In type-aware storage, higher-layers encode structural information about data by way of *pointers* and communicate it to the disk subsystem, enabling the disk to support semantics-aware optimizations and constraints. In context-aware storage, applications can communicate *application-data* and *application-I/O* relationships to the storage stack by way of logical *contexts*. In the slew of related work that aims at bridging the information-gap, our solution explores a unique design space of techniques that are powerful, simple, and easy to deploy. We have developed a number of case-studies using our two new abstractions, that demonstrate our claim.

In the rest of this section, we explain at more length, the motivation behind our work, existing solutions, and the effectiveness of our new abstractions.

1.1 Information-Gap in the Storage Stack

In a simple storage stack comprising applications, file systems, and the storage hardware, each of the layers have different kinds of information about the data they manage. Applications have extensive knowledge about the higher-level structures of data (e.g., a B-tree stored in a file) and their access characteristics. File systems know about the organization of data in the disk (e.g., the set of disk blocks belonging to a particular file), and disk systems contain hardware-specific information such as current head position, hardware RAID configuration, exact physical location of blocks (e.g., LBN to physical sector mapping systems) etc.

In today's storage stack, information available in each of these layers is not available to any of the other layers. If there is better exchange of information among the layers of the storage stack, a wide range of new functionality can be built. Here are some examples:

- High performance I/O-intensive applications such as databases and Web servers have long wanted a means for controlling the placement of their data on disk [73]. However, today's file systems do not provide mechanisms for higher level applications to communicate their desired data layout. For example, FFS-like file systems [52] such as Ext2 group files and directories based on generic heuristics and cannot tune their policies for individual applications. If file systems are aware of application-level access patterns and locality characteristics, they can perform informed placement decisions to optimize I/O.

- Many recent systems have looked at saving power by switching off a subset of disks in a large RAID array in such a way that applications can still function properly without the switched-off disks [87, 90]. These systems go to great complexity to identify the subset of data that is currently under use, yet these techniques are most often approximate and too coarse-grained. If the disk system has knowledge about the *working-sets* of data used by specific higher-level applications, they can do a much better job at such power optimizations by being more aggressive and more accurate.

- Reliability mechanisms (e.g., replication) at the storage system treat all blocks equally. However, from an application viewpoint, data blocks have varying importance. For example, meta-data blocks are more important than regular data as they impact accessibility of other data. Information about the semantic importance of data items can improve the effectiveness of reliability features.

1.2 Bridging the Information-Gap: Past Approaches

Utilizing application knowledge within the lower layers of a computer system has long been an attractive goal in computer systems research. Many systems have been proposed with this high-level goal of bridging the information gap between applications, the OS, and the hardware. Most existing research in this area can be classified under the following four categories.

Extensible systems. A very common way to bridge the information gap between applications and the system layers is to enable the system component to be dynamically extensible by the application. Extensible operating systems [10, 20, 26, 41, 47, 65] are examples of this type. There have also been proposals to extend storage hardware by enabling applications to dynamically download code [1, 62]. Extensible systems provide a lot of control to the application, but in the process, essentially ties them together. For applications to actually use such extensible layers, they need to have a reasonably intricate understanding of the system, thus making them complex to design.

Hint-based systems. Another approach that has been explored to solve the information gap problem is a more evolutionary one; provide specific primitives at the system level that the applications can use to convey information to the operating system. Informed prefetching [78] is an example of such a system. Researchers have also looked at the flip-side of the problem: provide information about the operating system to the application so that the application can make intelligent decisions [5, 31]. Most of these hint-based approaches are often tied to a specific kind of optimization or functionality. In other words, the information being transferred is designed with a particular purpose in mind. This in turn limits the flexibility of such a system because each new class of functionality may require yet another new primitive to be added to the interface.

Brand new interfaces. A large body of research that aims at bridging the information gap examine new interfaces between file systems and disk storage [11, 17, 18, 49]. The Object-based Storage Interface [53] is a classic example for this. Most of these interfaces are designed with some specific applications or scenarios in mind. For example, it is hard to implement a database in an object-based disk. Moreover, brand new interfaces require a complete revamp of existing infrastructures, and hence are unlikely to be deployed in the near future.

Inference-based systems. The final class of related work pertains to approaches that take the extreme viewpoint along the axis of being evolutionary and less intrusive. These systems attempt to achieve cross-layer awareness, but without explicitly communicating it from one layer to another [70, 71]. Gray-box systems [4] fall under this category. Although valuable from the viewpoint of being easily deployable and less intrusive, these approaches have their own limitations because they are heavily constrained in terms of not changing interfaces. This in many cases results in additional complexity, making it hard to reason about correctness while also limiting the usage of such inferred knowledge to less aggressive applications that can tolerate inaccuracy.

1.3 Our Approach

As described in Section 1.2, almost all existing solutions to bridge the information-gap have one or more of the following problems: (1) They are limited in scope such that they bridge the gap between just a sub-set of layers in the storage stack. (2) They require wide-scale modifications to existing infrastructures making their deployment too unlikely. (3) They are built with specific functionality in mind requiring explicit co-ordination between layers. In a multi-vendor setup, such coordination translates into industry-wide consensus on the interface, a standardization process that takes years.

Our approach to solve the problem of information-gap is to propagate *minimal* and *generic* information relating to data and I/O, from higher-level layers of the storage stack to the lowest-level (the storage hardware). We evolve two generic abstractions to encode *structural* and *operational* information available at the application-level and communicate it as part of I/O operations. Our first abstraction is *type-awareness*, which is to communicate **pointers** between disk blocks to the lower layers of the storage stack. Pointers establish relationships between disk blocks in a generic manner, and are maintained by layers such as file systems or databases. Our second abstraction is *context-aware storage*, which is to communicate higher-level **logical context** of I/O operations across the storage stack. For example, all I/O operations generated from a single user application can be grouped under the same logical context.

The following are the three key characteristics of our approach that differentiates our work from previous approaches.

1. The information being communicated from higher-level layers is already available at the corresponding layers (e.g., file systems already track block pointers), and hence communicating such information requires limited and straightforward modifications to existing infrastructures. More specifically, the modifications required to layers in our approach, are *implementation-level*. These modifications are much easier to make compared to the *design-level* modifications required with brand-new abstractions such as Object-based Storage [53].

2. By decoupling the *generation* of information at the higher layers from how the information is *used* at the lower layers, we obviate the need for explicit coordination between any two layers to support our abstractions. Our pointer or context information is not generated with any specific layer or functionality in mind.

3. Our abstractions extend end-to-end across the storage stack, (i.e., from user applications to the storage hardware), hence allowing a wide-range of interesting functionality in the different layers of the storage stack.

Next, we present our two abstractions in more detail and describe several case-studies that we built to demonstrate the generality and usefulness of our approach.

1.3.1 Type-Awareness

Pointers are the fundamental means by which modern file systems organize raw disk data into semantically-meaningful entities such as files and directories. Pointers define three things: (1) the semantic dependency between blocks (e.g., a data block is accessible only through a pointer from an inode block); (2) the logical grouping of blocks (e.g., blocks pointed to by the same indirect block are part of the same file or directory); and (3) the importance of a block (e.g., blocks with many outgoing pointers are important because they impact the accessibility of a large set of blocks).

Despite the rich semantic information inherently available through pointers, pointers are completely opaque to disk systems today. We propose the notion of a *type-safe disk* (TSD), a disk system that has knowledge of the pointer relationships between blocks. A TSD uses this knowledge in two key ways. First, semantic structure conveyed through pointers is used to enforce invariants on data access, providing better data integrity and security. For example, a TSD prevents access to an unallocated block. Second, a TSD can perform various semantics-aware optimizations that are difficult to provide in the current storage hierarchy [69, 70]. A TSD extends the traditional block-based read-write disk interface with three new primitives: block allocation, pointer creation, and pointer removal.

We demonstrate the utility of type-awareness through four prototype case studies.

A Capability Conscious Extended Storage System (ACCESS). In our first case-study, we show that a disk system can provide better data security by *constraining* data access to conform to implicit trust relationships conveyed through pointers. ACCESS (A Capability Conscious Extended Storage System) is a TSD prototype that provides an independent perimeter of security by constraining data access even when the operating system is compromised due to an attack.

Automatic Consistency Enforcing Disk (ACE-disk). In our second case study, ACE-disk, we show how a disk system can preserve the semantic consistency of data using pointers. ACE-disk automatically constructs dependency groups based on pointer operations and ensures atomic commit of blocks in resolved dependency groups. We developed a prototype ACE-disk and show how it preserves the consistency of an Ext2 file system.

A Discriminating Hierarchical Storage System. In the third case-study, we propose and evaluate a hierarchical storage system, DHIS, that is capable of discriminating between data with different access characteristics, and then customizing its layout and caching policies to each type. DHIS allows annotating pointers with a set of generic attributes that convey various properties such as data importance, access-patterns etc. We show via a prototype implementation that customizing policies to specific data requirements has significant performance benefits.

A Secure Deleting Disk System. The fourth case-study is secure deletion [30], a type-aware disk system prototype that automatically overwrites deleted blocks. When the last incoming pointer to a block is removed (making the block unreachable), our secure deleting disk overwrites the block to provide secure deletion. Disk level secure deletion is considered more secure, compared to existing software-level methods [69].

1.3.2 Context-Awareness

Our second abstraction is the concept of *Context-Aware I/O* (CAIO), a simple and generic way for applications to convey arbitrary information about their I/O behavior and relationships, without worrying about how the information will be used by the storage stack. In CAIO, an application-level *context* is propagated along with an I/O operation across the entire storage stack, in an end-to-end fashion. An application-level context is represented by one or more *context identifiers*. For example, a database application can have a unique identifier that it can propagate along with every I/O it generates, such that any storage layer can easily group all I/O generated by the database application.

Decoupling the generator and consumer of the context information leads to an interesting challenge: when the application could conceivably use more than one possible granularity of grouping I/O, how can it decide which one to use while being oblivious to how the grouping is interpreted by the lower level? For example, a database application can group the I/O requests it generates based on the database user, session, transaction, or query for which the I/O is issued; but the lower layers are oblivious to the granularity of the context. To solve this issue, contexts in CAIO are *hierarchical*. With hierarchical contexts, higher layers can encode multiple granularities of grouping, and the lower layers can decide which granularity is the best for the particular functionality that they provide.

We have implemented the CAIO framework in the Linux kernel and evaluated its usability. To illustrate the power and generality of this context abstraction, we prototyped and evaluated three case studies, described below:

Working-Set Identifier (WorkSIDE). Our first case study is an automatic working set identifier, *WorkSIDE*, which operates at the block-based storage hardware layer. WorkSIDE automatically tracks the data working set required for an application context to run to completion. WorkSIDE correlates contexts with the I/O and the corresponding data they access, thus obtaining a complete view of the entire set of data items that the particular application context requires. This working set can then be preloaded as appropriate in order to improve performance and availability, or to enable power optimizations.

Context-Aware Disk-Level Caching. Our second case study is a context-aware cache-placement algorithm within the disk that automatically tracks which application-level contexts exhibit sequential access pattern and avoids caching requests with that context. We demonstrate the usefulness of both of our case-studies using prototype implementations we built for the Linux kernel, and evaluate various workloads.

A Proportional-Share Disk Scheduler. In our third case-study, we built a context-based proportional-share disk scheduler which shows the flexibility of using resource shares associated with logical contexts, in a disk scheduler. We have implemented a prototype of our scheduler in the Linux kernel.

Overall, we find that our end-to-end abstractions enable a wide-range of functionality and optimizations in the different layers of the storage stack, while requiring minimal modifications to the storage interfaces.

1.4 Evaluation Methodology

We have implemented prototypes of both our abstractions and all the case-studies, in the Linux kernel 2.6.15. For evaluating disk-level functionality, we built our own software-level disk prototyping framework. Our framework operates as a pseudo device driver that interposes between the file system and the regular disk drivers. One key challenge in this prototyping environment is to ensure there is no performance interference between the host application and the processing at the pseudo driver layer. By careful use of kernel isolation techniques, we isolate the CPU and memory usage of the software prototype from the "host" applications, thus providing a very close approximation of an actual hardware prototype with its own processing and memory. We believe that this prototyping environment is valuable more generally for evaluating other kinds of functionality in the storage system.

1.5 Contributions

They key contributions of this dissertation are as follows:

- Formulation of the *pointer* abstraction and the design of the Type-Safe Disk interface that enables easy communication of higher-level pointers to the disk system.
- Design, implementation, and evaluation of four case-studies that demonstrate the security functionality and performance optimizations that type-awareness enables.
- Formulation of the *hierarchical context* abstraction and the Linux implementation of the context propagation infrastructure.
- Design, implementation, and evaluation of three case-studies to demonstrate the power and generality of the context abstraction.

- Implementation of a software-level framework to easily and accurately prototype disk-level functionality. This framework provides an interesting choice between hardware-level prototyping and entirely simulation-based prototyping.

1.6 Outline

The rest of this thesis is organized as follows.. Chapter 2 discusses some background information. In Chapter 3, we present the detail design, implementation, and evaluation of type-aware storage. In Chapters 4, 5, 6, and 7, we describe our four case-studies that use type-aware storage. Chapter 8 presents context-aware I/O. In Chapters 9, 10, and 11, we describe the three case-studies of context-aware I/O. In Chapter 12 we discuss related work, and we finally conclude in Chapter 13.

Chapter 2

Background

In this section, we discuss background information about the modern storage stack, large-scale storage systems, RAID levels, and file systems.

2.1 Modern Storage Stack

In the past file systems communicated directly with disks by using hardware-specific information such as tracks and sectors. The storage stack has evolved significantly since then. Figure 2.1 shows an example of the modern storage stack. Disk hardware information is virtualized through block-based interfaces such as SCSI and ATA. Layers such as RAID [59] or logical volume managers can exist beneath file systems, and they aggregate several independent disks. File systems are completely unaware of whether they are communicating with a single disk system or a RAID array. In today's storage stack, even a network can exist between file systems and the storage hardware [13, 64, 67, 75], and higher-level user applications are completely oblivous to these intermediate layers.

2.2 Large-scale Storage Systems

Large-scale storage systems today comprise diverse resources that include high processing power, hundreds of gigabytes of RAM, solid state storage media such as flash, and hundreds or even thousands of disks [16]. Modern storage systems run complex software to provide functionality such as reliability, fault-tolerance, and high performance I/O. One of the challenges in such storage systems is to effectively manage the wide range of resources to provide optimal performance and customizable features. However, despite the advancement in storage hardware, the interface used for communicating with hardware devices is still simple and narrow in most scenarios. For example, the SCSI interface supports just two main primitives, block `read` and `write`, resulting in the storage system being mostly oblivious to higher-level information. This makes efficient resource management within modern storage systems a difficult problem, as storage systems cannot discriminate between the different kinds of information they store.

Some existing systems try to work around this problem by exporting more information to higher-level software [18, 34]. For example, certain enterprise-class storage systems al-

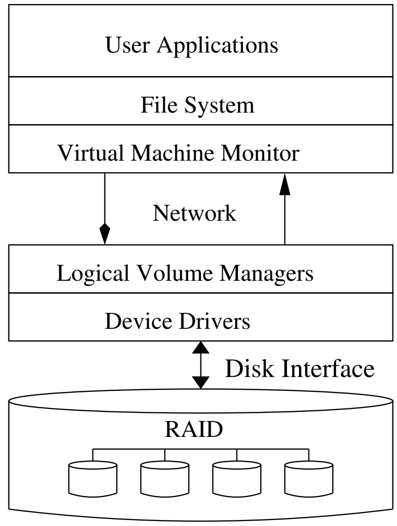


Figure 2.1: *Modern Storage Stack*

low higher-level software to choose the RAID level to use for a new volume, during its creation [35]. However, this requires that the file system or higher-level storage software be aware of the characteristics of each volume, which could be totally tied to the internal architecture of the specific storage systems. For example, a storage system could contain several fine-grained RAID levels, and devices such as NVRAM and solid state memory. Storage architectures could also be different across vendors and models, and it may be cumbersome to customize file systems for specific storage systems. Moreover, the abstraction of a volume is in most cases too coarse-grained to express difference in access characteristics across files.

2.3 RAID levels

Redundant Array of Independent Disks (RAID) is one of the most common ways used to improve storage system performance and reliability [36, 59]. There are several configurations of RAID (commonly referred to as *RAID levels*) that are used in practice. Each of these configurations have their own performance, reliability, and cost characteristics. In this section, we describe three most commonly used RAID levels.

RAID0 performs plain striping across several disks without any redundancy and hence it has the lowest reliability level among the three. However, in terms of performance, RAID0 is good for sequential and random read-write workloads. This is mainly because I/O operations get parallelized across the individual disks when data is striped. In terms of cost per gigabyte, RAID0 is the cheapest as there is no redundancy and the storage capacity is the sum of the individual disk capacities.

RAID1 mirrors data across two disks. As two disks contain identical data at all times, data reliability is better as it can tolerate a single disk failure. In terms of performance, RAID1 has similar characteristics for both sequential and random I/O. Reads are faster than writes as reads can be parallized across the two disks. Write speed is in tune with that of a single disk, because for every write, both disks have to be updated, but in parallel. RAID1 has the highest cost per gigabyte as the total capacity of the drives is halved due to mirroring.

RAID5 stripes both data and parity information across three or more drives. In principle it is similar to having a single dedicated parity drive, but parity blocks are distributed across all drives RAID5 can recover from single disk failures and hence has comparable reliability to RAID1. Read performance in RAID5 is similar to that of RAID0. However, for small random writes RAID5 performs poorly. This is because for small writes that do not span a complete stripe, computation of new parity involves reading the old contents of the data block and the parity block. In terms of cost per gigabyte, RAID5 is the second best among the three, as there is a single parity block for a stripe.

2.4 Overview of File Systems

Several applications need to store data persistently on secondary storage disks. Storage software such as file systems and databases provide a generic interface to access storage devices and maintain their own structures to track abstractions. For example, each file system has its own on-disk layout. In this section, we provide a background of file systems in general and about the layout of the Ext2 file system in particular. We also discuss briefly a few other common storage structures that software use to manage data on disk.

File systems abstract raw disk blocks into logical entities such as file and directories. To track the set of blocks that constitute a logical file or directory, a file system uses various forms of *meta-data*; such meta-data can be broadly classified into directories, file-specific meta-data, and structures required for free-space management. Directories link logical file identifiers to file specific meta-data. File-specific meta-data contains the file attributes and links to the actual data blocks. Allocation structures include bitmaps and free-lists that required for managing disk space. In common Unix file systems that follow the semantics of the Berkeley Fast File System (FFS) [52], per-file meta-data objects are called *inodes*.

2.4.1 The Layout of the Ext2 File System

The Ext2 file system which has its roots in BSD's FFS, groups together a fixed number of sequential blocks into a block group and the file system is managed as a series of block groups. This is done to keep related blocks together. Each block group contains a copy of the super block, inode and block allocation data-structures, and the inode blocks. The inode table is a contiguous array of blocks in the block group that contains on-disk inodes. The number of inodes and their location are statically determined during the *mkfs* operation. Each inode block can contain several inodes. Each inode inside a block is treated as an *allocatable* unit, and bitmaps keeps track of allocated and free inodes within a block group.

2.4.2 File System Consistency

Today's block-based disks export a flat array-like abstraction of fixed size blocks. To manage data in the form of groups (e.g., a file) and to provide the notion of hierarchy (such as directories), they need to manage pointers between blocks. Such pointers are vital entities in storage and in most cases they impact the accessibility of the data. For example, when an inode block is lost, all data pertaining to the corresponding files become unreachable and hence inaccessible. More importantly, the *consistency* of these pointers determines to a large extent the semantic consistency of the information stored in a disk. For example, during a `rename` operation in Ext2, a directory entry (which is a pointer to an inode block) in a directory block is removed and added in another directory block. If the system crashes after the removal operation is done, a file becomes inaccessible even though its data items are intact. While complex storage software maintain strong forms of consistency such as the consistency between the size field in an inode and the actual file size, mere pointer consistency is sufficient in most cases. For example, if all pointers from an inode are consistent, the size field can be re-constructed by just looking at the set of pointers. In this work, we focus on ensuring pointer consistency at the disk level.

Chapter 3

Type-Aware Storage Infrastructure

Type-safety is a well explored concept in the field of programming languages, with proven benefits such as controlled access to memory. We propose to extend the property of type-awareness and type-safety to the disk subsystem, and show that it can significantly improve the security and functionality of the disk subsystem. Specifically, we advocate regulating access to disk blocks to conform to well-defined rules, that are understood and enforced by the disk itself. In building this, we leverage the fact that the semantics of most file systems today can be broadly classified into two categories: raw data blocks, and *pointers* or references that implement logical relationships between data blocks (for example, dentries-inodes and inodes-data blocks). We define a *type-aware* disk as one that can differentiate between these two distinct types of information it stores. Once a disk has this information, it can exploit this knowledge to provide better functionality. We believe that this simple type-awareness could be a significant source of semantic information that can bridge the semantic gap between file systems and storage devices. Although several existing research projects like Object-store Disks (OSD) explore alternatives to bridge this gap, we believe that a *data-pointer* abstraction is the right interface that a disk should provide to file systems. A disk that is type-aware can *enforce* type safety by limiting block accesses to only the legal set of pointers, thus preventing arbitrary block dereferencing. We call such a disk a *type-safe disk* (TSD).

TSDs require a few changes to the current block-based interface. First, like any other type-safe system, allocation and deallocation has to be under the control of the disk ssyste. By performing block allocation and de-allocation, a TSD frees the file system from the need for free-space management. Similar in spirit to type-safe programming languages, a TSD also exploits its pointer awareness to perform automatic garbage collection of unused blocks; blocks which have no pointers pointing to them are reclaimed automatically, thus freeing file systems of the need to track reference counts for blocks in many cases.

In this chapter we present in more detail, our type-aware storage abstraction, and four case-studies that we built to show the usefulness of our abstraction.

This chapter is organized as follows. In Section 3.1 we discuss the utility of pointer information at the disk. Section 3.2 discusses the design and implementation of the basic TSD framework. In Section 3.3 we describe file system support for TSDs. In Section 3.4 we present the software-level disk prototyping environment that we built to evaluate the idea of TSDs and all our case-studies. We present the evaluation of our prototype implementation of TSD in Section 3.6.

3.1 Motivation

In this section we present an extended motivation.

Pointers as a proxy for data semantics The inter-linkage between blocks conveys rich semantic information about the structure imposed on the data by higher layers. Most modern file systems and database systems make extensive use of pointers to organize disk blocks. For example, in a typical file system, directory blocks logically point to inode blocks which in turn point to indirect blocks and regular data blocks. Blocks pointed to by the same pointer block are often semantically related (e.g., they belong to the same file or directory). Pointers also define reachability: if an inode block is corrupt, the file system cannot access any of the data blocks it points to. Thus, pointers convey information about which blocks impact the availability of the file system to various degrees. Database systems are very similar in their usage of pointers. They have B-tree indexes that contain on-disk pointers, and their extent maps track the set of blocks belonging to a table or index.

In addition to being passively aware of pointer relationships, a type-safe disk takes it one step further. It actively enforces invariants on data access based on the pointer knowledge it has. This feature of a TSD enables independent verification of file system operations; more specifically, it can provide an additional perimeter of security and integrity in the case of buggy file systems or a compromised OS. As we show in Section 4, a type-safe disk can limit the damage caused to stored data, even by an attacker with root privileges. We believe this active nature of control and enforcement possible with the pointer abstraction makes it powerful compared to other more passive information-based interfaces.

Pointers thus present a simple but general way of capturing application semantics. By aligning with the core abstraction used by higher-level application designs, a TSD has the potential to enable on-disk functionality that exploits data semantics. In the next subsection, we list a few examples of new functionality (some proposed in previous work in the context of alternative approaches) that TSDs enable.

Applications There are several possible uses of TSDs.

Selective Data Replication Since TSDs are capable of differentiating data and pointers, they can identify metadata blocks as those blocks that contain outgoing pointers and replicate them to a higher degree, or distribute them evenly across all the disks. This could provide graceful degradation of availability as provided by D-GRAID [70].

Data colocation Using the knowledge of pointers, a TSD can co-locate blocks along with their reference blocks (blocks that point to them). In general, blocks will be accessed just after their pointer blocks are accessed, and hence there would be better locality during access.

Intelligent Prefetching TSDs can perform intelligent prefetching of data because of the pointer information. When a pointer block is accessed, a TSD can prefetch the data blocks pointed to by it, and store it in the on-disk buffers for improved read performance.

Disk-level security TSDs can provide new security properties using the pointer knowledge by enforcing *implicit* capabilities. We discuss this in detail in Section 4.

Secure deletion TSDs can perform automatic secure deletion of deleted blocks by tracking block liveness using pointer knowledge. We describe this in detail in Section 7.

3.2 Type-Safety at the Disk Level

Having pointer information inside the disk system enables enforcement of interesting constraints on data access. For example, a TSD allows access to only those blocks that are reachable through some pointer path. TSDs manage block allocations and enforce that every block must be allocated in the context of an existing pointer path, thus preventing allocated blocks from becoming unreachable. More interestingly TSDs enable disk-level enforcement of much richer constraints for data security as described in our case study in section 4.

Enforcing such access constraints based on pointer relationships between blocks is a restricted form of *type-safety*, a well-known concept in the field of programming languages. The type information that a TSD exploits, however, is narrower in scope: TSDs just differentiate between normal data and pointers.

We now detail the TSD interface, its operation, and our prototype implementation. Figure 3.1 shows the architectural differences between normal disks and a TSD.

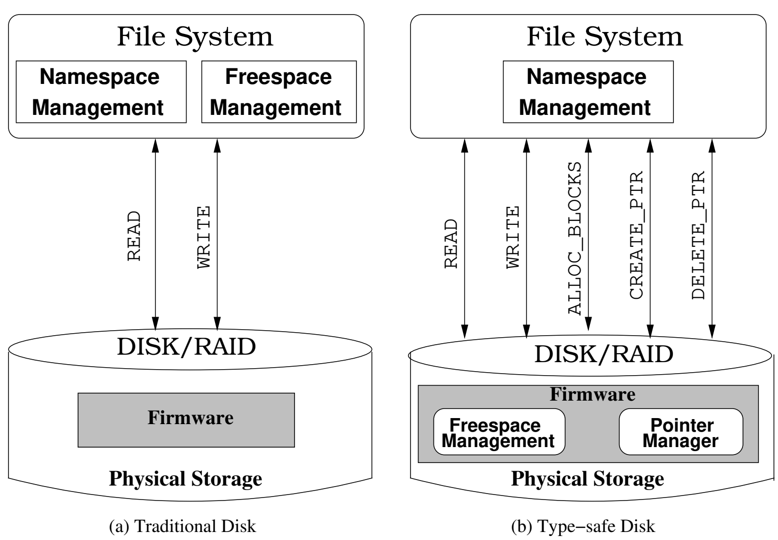


Figure 3.1: *Comparison of traditional disks vs. type-safe disks*

3.2.1 Disk API

A type-safe disk exports the following primitives, in addition to the basic block-based API:

- SET_BLOCKSIZE($Size$): Sets the file system block size in bytes.
- ALLOC_BLOCKS($Ref, Hint, Count$): Allocates $Count$ number of new file system blocks from the disk-maintained free block list, and creates pointers to the allocated blocks, from block Ref. Allocated blocks need not be contiguous. Ref must be a valid block number that was previously allocated. $Hint$ is the block number closest to which the new blocks should be allocated. $Hint$ can be NULL, which means the disk can choose the new block totally at its own discretion. Returns an array of addresses of the newly allocated blocks, or NULL if there are not enough free blocks on the device.
- ALLOC_CONTIG_BLOCKS($Ref, Hint, Count$): Follows the same semantics as ALLOC_BLOCKS, except that it allocates $Count$ number of contiguous blocks if available.
- CREATE_PTR($Src, Dest$): Creates a pointer from block Src to block $Dest$. Both Src and $Dest$ must be previously allocated. Returns success or failure.
- DELETE_PTR($Src, Dest$): Deletes a pointer from block Src that points to block $Dest$. Semantics similar to CREATE_PTR.
- GET_FREE: Returns the number of free blocks left.

3.2.2 Managing Block Pointers

A TSD needs to maintain internal data-structures to keep track of all pointers between blocks. It maintains a pointer tracking table called PTABLE that stores the set of all pointers. The PTABLE is indexed by the source block number and each table entry contains the list of destination block numbers. A new PTABLE entry is added every time a pointer is created. Based on pointer information, TSD disk blocks are classified into three kinds: (a) *Reference blocks*: blocks with both incoming and outgoing pointers (such as inode blocks). (b) *Data blocks*: blocks without any outgoing pointers but just incoming pointers. (c) *Root blocks*: a pre-determined set of blocks that contain just outgoing pointers but not incoming pointers. Root blocks are never allocated or freed, and they are statically determined by the disk. Root blocks are used for storing boot information or the primary metadata block of file systems (e.g., the Ext2 super block).

3.2.3 Free-Space Management

To perform free-space management at the disk level, we track live and free blocks. A TSD internally maintains an allocation bitmap, ALLOC-BITMAP, containing one bit for every logical unit of data maintained by the higher level software (e.g., a file system block). The size of a logical unit is set by the upper-level software through the SET_BLOCKSIZE disk primitive. When a new block need to be allocated, the TSD can choose a free block closest to the hint block number passed by the caller. Since the TSD can exploit the low level knowledge it has, it chooses a block number which requires the least access time from the hint block.

TSDs use the knowledge of block liveness (a block is defined to be dead if it has no incoming pointers) to perform garbage collection. Unlike traditional garbage collection systems in programming languages, garbage collection in TSD happens *synchronously* during a particular DELETE_PTR call which deletes the last incoming pointer to a block. A TSD maintains a reference count table, RTABLE, to speed up garbage collection. The reference count of a block gets incremented every time a new incoming pointer is created and is decremented during pointer deletions. When the reference count of a block drops to zero during a DELETE_PTR call, the block is marked free immediately. A TSD performs garbage collection one block at a time as opposed to performing cascading deletes. Garbage collection of reference blocks with outgoing pointers is prevented by disallowing deletion of the last pointer to a reference block before all outgoing pointers in it are deleted.

3.2.4 Consistency

As TSDs maintain separate pointer information, TSD pointers could become inconsistent with the file system pointers during system crashes. Therefore, upon a system crash, the consistency mechanism of the file system is triggered which checks file system pointers against TSD pointers and first fixes any inconsistencies between both. It then performs a regular scan of the file system to fix file system inconsistencies and update the TSD pointers appropriately. For example, if the consistency mechanism creates a new inode pointer to fix an inconsistency, it also calls the CREATE_PTR primitive to update the TSD internal pointers. Alternatively, we can obviate the need for consistency mechanisms by just modifying file systems to use TSD pointers instead of maintaining their own copy in their meta-data. However, this involves wide-scale modifications to the file system.

File system integrity checkers such as `fsck` for TSDs have to run in a privileged mode so that they can perform a scan of the disk without being subjected to the constraints enforced by TSDs. This privileged mode can use a special administrative interface that overrides TSD constraints and provides direct access to the TSD pointer management data-structures.

Block corruption When a block containing TSD-maintained pointer data-structures gets corrupted the pointer information has to be recovered, as the data blocks pertaining to the pointers could still be reachable through the file system meta-data. Block corruption can be detected using well-known methods such as checksumming. Upon detection, the TSD notifies the file system, which recreates the lost pointers from its meta-data.

3.3 File System Support

We now describe how a file system needs to be modified to use a TSD. We first describe the general modifications required to make any file system work with a TSD. Next, we describe our modifications to two file systems, Linux Ext2 and VFAT, to use our framework.

Since TSDs perform free-space management at the disk-level, file systems using TSD are freed from the complexity of allocation algorithms, and tracking free block bitmaps and other related meta-data. However, file systems now need to call the disk API to perform allocations, pointer management, and getting the free blocks count. The following are the general modifications required to existing file systems to support type-safe disks:

1. The `mkfs` program should set the file system block size using the SET_BLOCKSIZE primitive, and store the primary meta-data block of the file system (e.g., the Ext2 super block) in one of the TSD root blocks. Note that the TSD root blocks are a designated set of well-known blocks known to the file system.

2. The free-space management sub-system should be eliminated from the file system, and TSD API should be used for block allocations. The file system routine that estimates free-space, should call the GET_FREE disk API, instead of consulting its own allocation structures.

3. Whenever file systems add new pointers to their meta-data, CREATE_PTR disk primitive should be called to create a TSD pointer. Similarly, the DELETE_PTR primitive has to be called when pointers are removed from the file system.

In the next two sub-sections we describe the modifications that we made to the Ext2 and the VFAT file systems under Linux, to support type-safe disks.

3.3.1 Ext2TSD

We modified the Linux Ext2 file system to support type-safe disks; we call the modified file system *Ext2TSD*. The Ext2 file system groups together a fixed number of sequential blocks into a block group and the file system is managed as a series of block groups. This is done to keep related blocks together. Each block group contains a copy of the super block, inode and block allocation data-structures, and the inode blocks. The inode table is a contiguous array of blocks in the block group that contain on-disk inodes.

To modify Ext2 to support TSDs, we removed the notion of block groups from Ext2. Since allocations and de-allocations are done by using the disk API, the file system need not group blocks based on their order. However, to perform easy inode allocation in tune with Ext2, we maintain inode groups which we call ISEGMENTS. Each isegment contains a segment descriptor that has an inode bitmap to track the number of free inodes in that isegment. The inode allocation algorithm of Ext2TSD is same as that of Ext2. The `mkfs` user program of Ext2TSD writes the super block, and allocates the inode segment descriptor blocks, and inode tables using the allocation API of the disk. It also creates pointers from the super block to all blocks containing isegment descriptors and inodes tables.

The organization of file data in Ext2TSD follows the same structure as Ext2. When a new file data or indirect block is allocated, Ext2TSD calls ALLOC_BLOCKS with the corresponding inode block or the indirect block as the reference block. While truncating a file, Ext2TSD just deletes the pointers in the indirect block branches in the right order such that all outgoing pointers from the parent block to its child blocks are deleted before deleting the incoming pointer to the parent block. Thus blocks belonging to truncated or deleted files are automatically reclaimed by the disk.

In the Ext2 file system, each directory entry contains the inode number for the corresponding file or directory. This is a logical pointer relationship between the directory block and the inode block. In our implementation of Ext2TSD, we create physical pointers between a directory block and the inode blocks corresponding to the inode numbers contained in every directory entry in the directory block. Modifying the Ext2 file system to support TSD was relatively simple. It took 8 days for us to build Ext2TSD starting from a vanilla Ext2 file system. We removed 538 lines of code from Ext2 which are mostly the code required for block allocation and bitmap management. We added 90 lines of new kernel code and modified 836 lines of existing code.

3.3.2 VFATTSD

The next file system we consider is VFAT, a file system with origins in Windows. Specifically, we consider the Linux implementation of VFAT. We chose to modify VFAT to support TSDs because it is sufficiently different in architecture from Ext2 and hence shows the generality of the pointer level abstraction provided by TSDs. We call our modified file system *VFATTSD*.

The VFAT file system contains an on-disk structure called the File Allocation Table (FAT). The FAT is a contiguous set of blocks in which each entry contains the logical block number of the next block of a file or a directory. To get the next block number of a file, the file system consults the FAT entries corresponding to the previous block of the file. Each file or directory's first block is stored as part of the directory entry in the corresponding directory block. The FAT entry corresponding to the last block of a file contains an `EOF` marker. VFAT tracks free blocks by having a special marker in the FAT entry corresponding to the blocks.

In the context of TSDs, we need not use the FAT to track free blocks. All block allocations are done using the allocation API provided by a TSD. The `mkfs` file system creation program allocates and writes the FAT blocks using the disk API. Modifying the VFAT file system to support TSDs was substantially simpler compared to Ext2, as VFAT does not manage data blocks hierarchically. We had to maintain substantially lesser number of pointers.

In VFAT, we created pointers from each directory block to all blocks belonging to files which have their directory entries in the directory block. Each FAT block points to the block numbers contained in the entries present within. The TSD therefore tracks all blocks belonging to files in the same directory block. Also, all the directory blocks and the FAT blocks contain outgoing pointers. The disk can track the set of all metadata blocks present in the file system by just checking if a block is a data block or a reference block.

Modifying the VFAT file system to support TSD was relatively straightforward. It took 4 days for us to build VFATTSD from the VFAT file system. We added 83 lines of code, modified 26 lines of code, and deleted 71 lines of code. The deleted code belonged to the free space management component of VFAT.

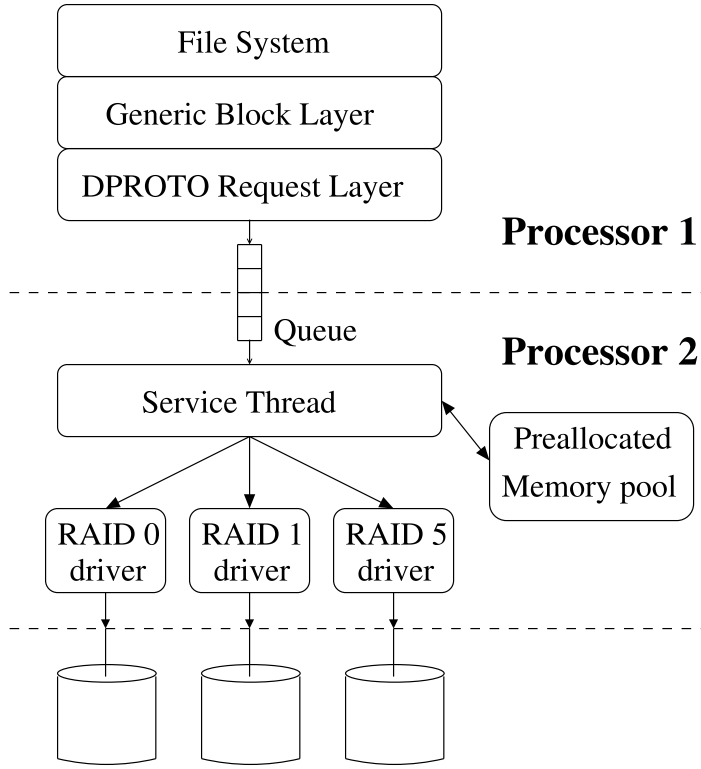


Figure 3.2: *DPROTO Architecture*

3.4 A Software-Level Disk Prototyping Framework

In this section, we describe our generic disk functionality prototyping framework, DPROTO, that we built for the Linux kernel 2.6.15.

We developed DPROTO as a pseudo-device driver that stacks on top of one or more lower-level disk or software RAID drivers, in a single machine. One of the main challanges in developing DPROTO is isolating the resources consumed by components that are supposed to go inside the disk firmware if it were a real implementation. For example, if the functionality being prototyped is a disk-level data compression technique, the part of DPROTO that performs compression has to consume resources that are completely isolated from that used by applications and file systems, which is difficult in a single machine setup.

While developing DPROTO we aimed at isolating key resources, CPU and memory, between disk-level functionality and higher-level applications. For CPU isolation, we use a multiprocessor setup and ensure that disk-level functionality always gets executed in an isolated processor. For memory isolation, we implemented an isolated preallocated memory pool and ensured that disk functionality never accesses memory beyond the preallocated range.

Figure 3.2 shows the architecture of DPROTO. We implemented the pseudo-device driver as two layer, the upper layer running in the context of the file system, and the lower layer running as a separate thread bound to an isolated CPU. Disk I/O requests generated from the file system reach the upper layer of DPROTO, which adds the request to a shared queue. The lower layer services requests from the queue and eventually passes it down to physical storage. Any disk-level functionality such as compression would be handled by the lower-level service thread and hence runs in an isolated CPU. All memory allocations done by both layers of DPROTO use the preallocated memory pool. Therefore, DPROTO requires specifying the total memory requirement for a given functionality before hand.

To test the performance of a disk-level functionality prototyped using DPROTO, the comparison reference can be run with one processor disabled and with the appropriate size of memory preallocated. For example, if a compression disk system is compared to a regular disk system for a particular workload, the regular disk run of the workload has to be done with one processor disabled and the preallocated memory equal to the memory requirement of the compression disk. With this procedure, the comparison becomes fair and closely represents the results of a real implementation.

Our implementation of DPROTO had 5,790 lines of new kernel code and 350 lines of user-level code.

3.5 TSD Implementation

We implemented a prototype TSD using our DPROTO software-level disk prototyping framework, in the Linux kernel 2.6.15. It contains 3,108 lines of kernel code. The TSD layer receives all block requests, and redirects the common read and write requests to the lower level device driver. The additional primitives required for operations such as block allocation and pointer management are implemented as driver `ioctls`.

We implemented PTABLE and RTABLE as in-memory hash tables which gets written out to disk at regular intervals of time through an asynchronous commit thread. In implementing the RTABLE, we add an optimization to reduce the number of entries maintained in the hash table. We add only those blocks whose reference count is greater than one. A block which is allocated and which does not have an entry in the RTABLE is deemed to have a reference count of one and an unallocated block (as indicated by the ALLOC_BITMAP) is deemed to have a reference count of zero. This significantly reduces the size of our RTABLE, because most disk blocks have reference counts of zero or one (e.g., all data blocks have reference counts zero or one).

Memory usage. In our prototype implementation we maintained all TSD data-structures in memory. The space overheads associated with TSD pointer tracking and free-space management is directly related to the number of file system blocks on disk. We found that the TSD pointer meta-data per file system block will be close to 20 bytes (with an average of one incoming pointer per block). Assuming a file system block size of 4KB, the total space overheads for TSDs totals upto 0.5% of the disk size. In a real firmware-level implementation of TSDs, the entire meta-data need not be maintained in memory. At any given time, the working-set of TSD pointers is limited to the directories and files being accessed at any given time. Hence, we believe that it would be sufficient if a fraction of the TSD meta-data (about 10%) is cached in memory, and the rest of the meta-data can be stored on secondary storage.

3.6 Evaluation

We evaluated the performance of our prototype TSD framework in the context of Ext2TSD. We ran general-purpose workloads and also micro-benchmarks on our prototype and compared them with unmodified Ext2 file system on a regular disk. This section is organized as follows: first we talk about our test platform, configurations, and procedures. Next, we analyse the performance of the TSD framework with the Ext2TSD file system.

3.6.1 Test infrastructure

We conducted all tests on a 2.8GHz Xeon with 1GB RAM, and a 250GB, LSILogic SCSI disk. We used Fedora Core 6, running a vanilla Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it. We recorded disk statistics from `/proc/diskstats` for our test disk. We analysed the following detailed disk-usage statistics while interpreting the results: the number of read I/O requests (`rio`), number of write I/O requests (`wio`), number of sectors read (`rsect`), number of sectors written (`wsect`), number of read requests merged (`rmerge`), number of write requests merged (`wmerge`), total time taken for read requests (`ruse`), and the total time taken for write requests (`wuse`).

3.6.2 Benchmarks and configurations

Postmark We used Postmark v1.5 to generate an I/O-intensive workload. Postmark stresses the file system by performing a series of operations such as directory lookups, creations, and deletions on small files [42]. Postmark is typically configured by specifying a number of initial files, and a fixed number of *transactions*. Postmark then creates the initial pool of files, performs the fixed number of transactions, and removes any left over files.

Kernel Compile To simulate a relatively CPU-intensive user workload, we compiled the Linux kernel source code. We used a vanilla Linux 2.6.15 kernel, and analyzed the overheads of Ext2TSD, for the `make oldconfig` and `make` operations combined.

Sprite LFS Benchmarks To isolate the overheads of individual file system operations, we ran the entire suite of Sprite LFS benchmarks [63]. The Sprite LFS benchmarks contains two sets of workloads, for meta-data and data operations. The first set deals with small files and tests, file creation, read, and file deletion. The second set operates on large files and performs sequential and random reads and writes.

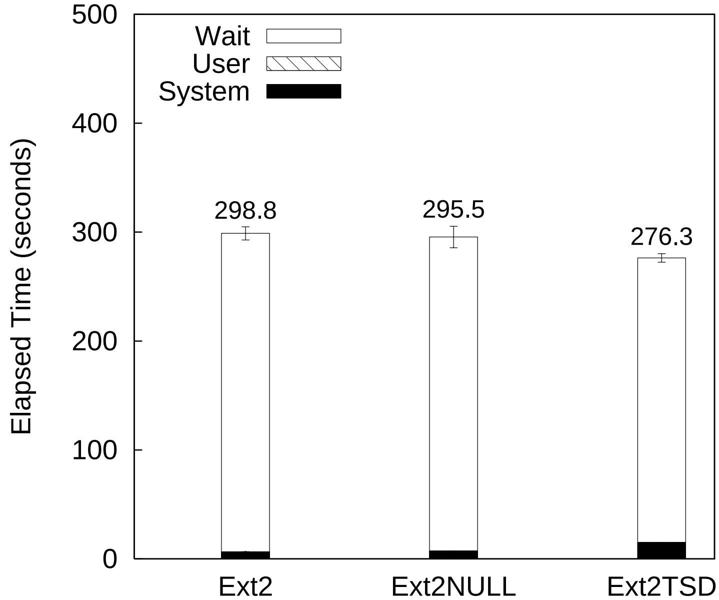


Figure 3.3: *Postmark Results: 10,000 files, sizes 100KB to 200KB, 10,000 transactions. Ext2NULL indicates the results for regular Ext2 over a NULL pseudo-device driver.*

3.6.3 Postmark Results

We ran the Postmark benchmark on three setups: (1) regular Ext2 over a regular disk, (2) regular Ext2 on DPROTO, and (3) Ext2TSD over our implementation of TSD. We used configured Postmark with two different configurations. In the first configuration, we used 10,000 files with sizes ranging from 100KB to 200KB, and 10,000 transactions. Figure 3.3 shows the overheads of DPROTO and the TSD infrastructure for this configuration. As evident from the figure, Ext2 over our prototyping infrastructure DPROTO had negligible overheads compared to Ext2 over a regular disk. However Ext2TSD ran 7% faster than regular Ext2 inspite of a 1.3 times increase in system time. The increase in system time is because of the device `ioctls` that Ext2TSD calls for the pointer operations. From the kernel disk I/O statistics, we found that the 10% decrease in wait time for Ext2TSD compared to regular Ext2 is caused by more requests getting merged at the device driver layer. This is because, block allocation is performed by TSDs in the case of Ext2TSD, and there was better spatial locality compared to regular Ext2.

Figure 3.4 shows the results for a different configuration of Postmark. For this we used 1000 files with sizes ranging from 1MB to 3MB, and performed 5000 transactions. In this configuration, Ext2TSD had an elapsed time overhead of 5.7% compared to regular Ext2. The system time overhead was 1.9 times and wait time was 5% lesser than regular Ext2. This shows that for larger files, the savings in I/O time because of better spatial locality is lesser compared to smaller files.

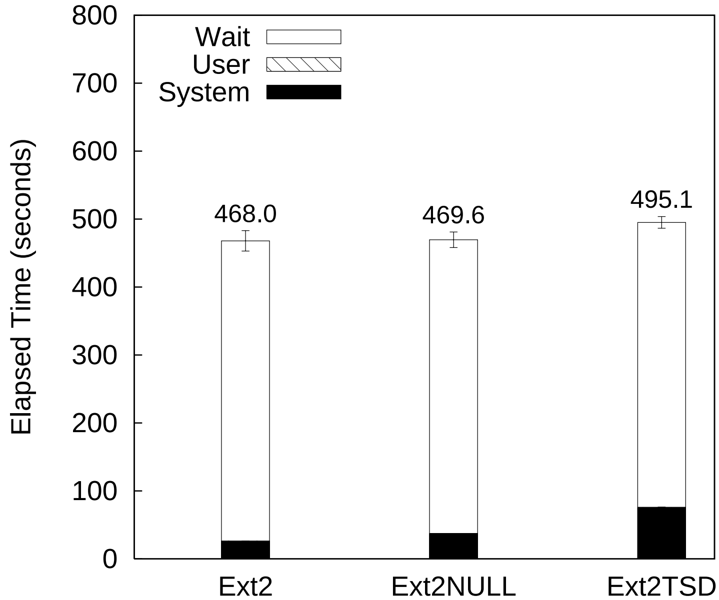


Figure 3.4: *Postmark Results: 1000 files, sizes 1MB to 3MB, 5000 transactions. Ext2NULL indicates the results for regular Ext2 over a NULL pseudo-device driver.*

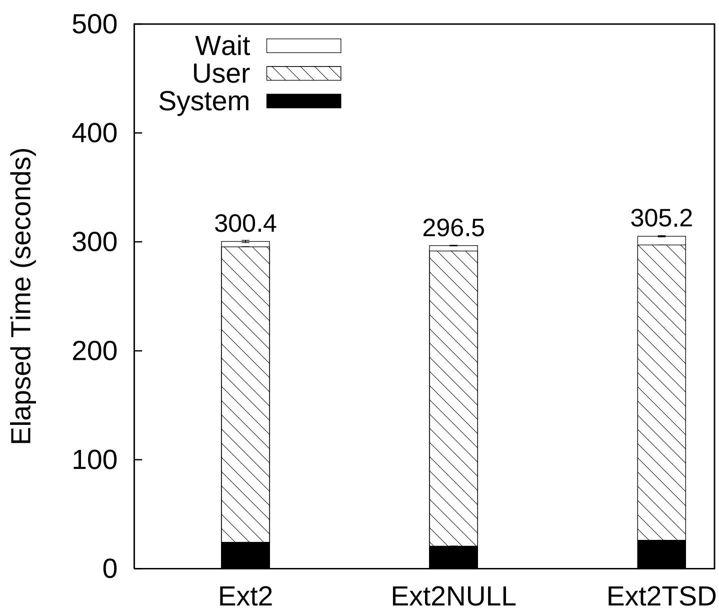


Figure 3.5: *Kernel Compile Results. Ext2NULL indicates the results for regular Ext2 over a NULL pseudo-device driver.*

3.6.4 Kernel Compile Results

Results for the kernel compilation benchmark is shown in Figure 3.5. Ext2TSD had a small elapsed time overhead of 1.5% compared to regular Ext2. This was caused by a 7% increase in system time and 60% increase in wait time. The system time increase in this case is smaller compared to the Postmark results because kernel compile is a predominantly CPU-intensive workload and hence has much lesser number of pointer operations. The wait time increase is because the main compilation thread waits for the DPROTO disk thread to complete pointer operations. The wait time increase is more pronounced here because the time interval between I/O is larger than that of Postmark.

3.6.5 Sprite LFS Benchmark Results

We ran the entire suite of Sprite LFS benchmarks on Ext2 over a regular disk, and Ext2TSD over our prototype TSD.

Meta-data benchmarks To generate a small file creation workload, we created 1,000,000 files, with size 4KB each, in 1,000 sub-directories. For reads, we remounted the file system and read all the 1,000,000 files we created. For deletes, we unlinked all files.

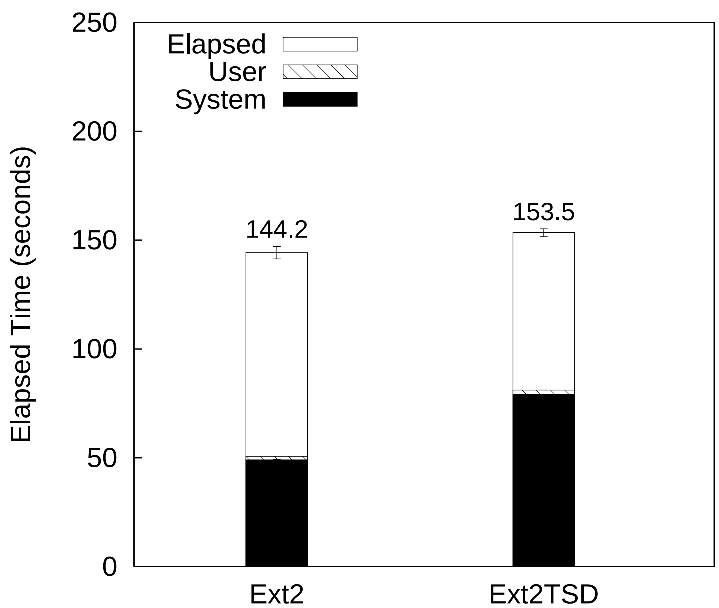


Figure 3.6: *Sprite LFS benchmarks: Create results*

Figure 3.6 shows the overheads of Ext2TSD. Ext2TSD had an elapsed time overhead of 6.4% compared to regular Ext2. This is because of a 61% increase in system time. The system time increase is because of the pointer operations as this workload consists of intensive meta-data write operations. The wait time mainly caused by I/O, reduced by 22% because the TSD allocation policy is favorable for small files.

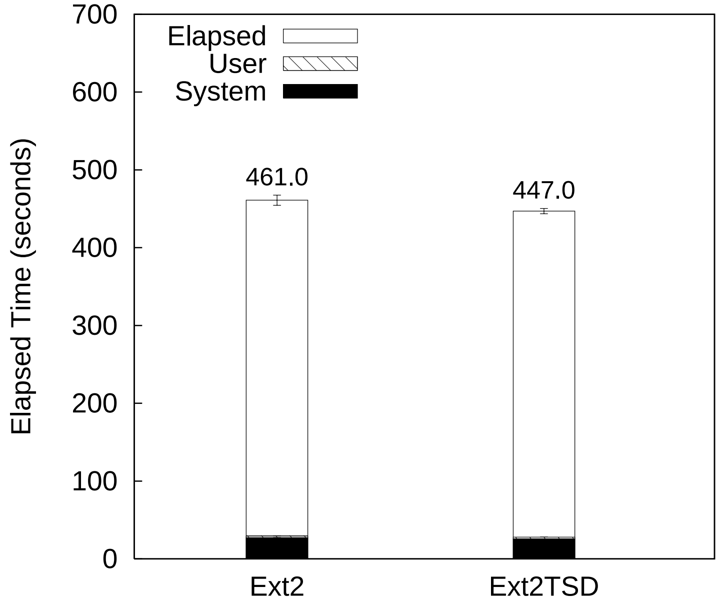


Figure 3.7: *Sprite LFS benchmarks: Read results*

Figure 3.7 shows the results of the read workload. Ext2TSD performed 3% better than regular Ext2. The system time reduced by 5% because of two reasons. First, there are no pointer operations in a read workload. Second, the isolation technique in DPROTO offloads part of call stack of I/O operations such as lower level SCSI driver calls, to the DPROTO disk thread.

The delete workload results shown in Figure 3.8 shows that the elapsed time overhead of Ext2TSD is 23% compared to regular Ext2. This is caused because of a 2.1 times increase in system time. This increase is because of a large number of pointer deletion operations happening within a short period of time.

Overall, even under extremely meta-data intensive workloads, the elapsed time overheads are moderate. In most common environments such meta-data intensive workloads are unlikely.

Data benchmarks For generating Sprite LFS data benchmark workloads, we used a large file of size 4GB. For random workloads we performed 10,000 random 4K reads or writes. To eliminate cache effects, we generated a duplicate free list of random page numbers. For sequential workloads, we performed 1,000,000 sequential 4K reads on the file.

Figure 3.9 shows the results for the random read workload. Ext2TSD had no visible overheads for this. As this is a read workload, it generated no pointer operations. For random write, as shown in Figure 3.10, Ext2TSD had an elapsed time overhead of 6.7%. This is mainly caused by a 6.7% increase in wait time. The wait time increase is because the main benchmark thread had to wait for the DPROTO disk thread to service pointer operations.

Figure 3.11 shows the results for sequential read. The difference in elapsed time

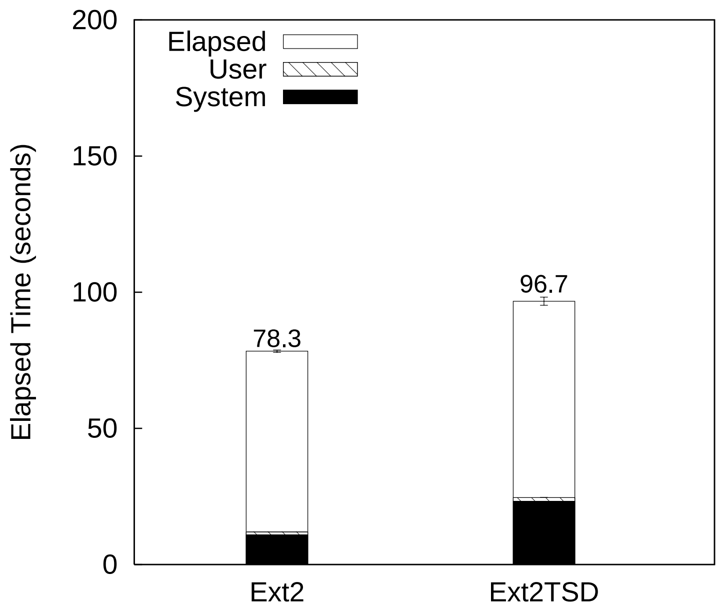


Figure 3.8: *Sprite LFS benchmarks: Unlink results*

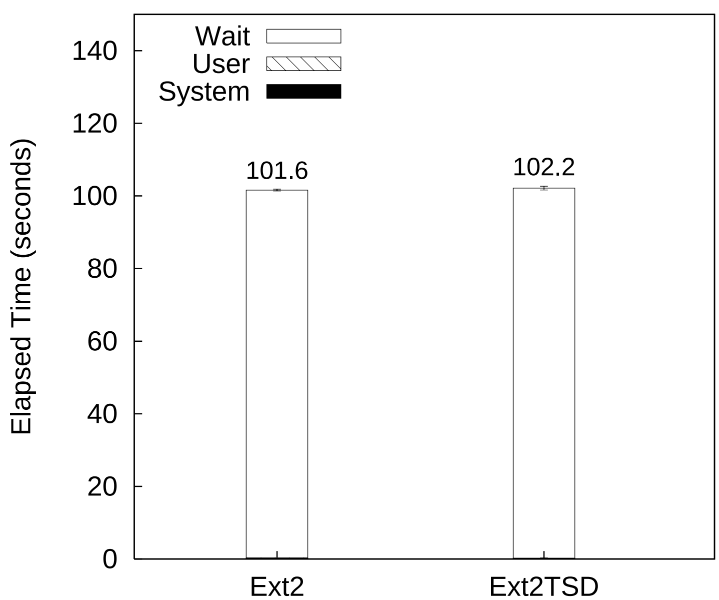


Figure 3.9: *Sprite LFS benchmarks: Random read results*

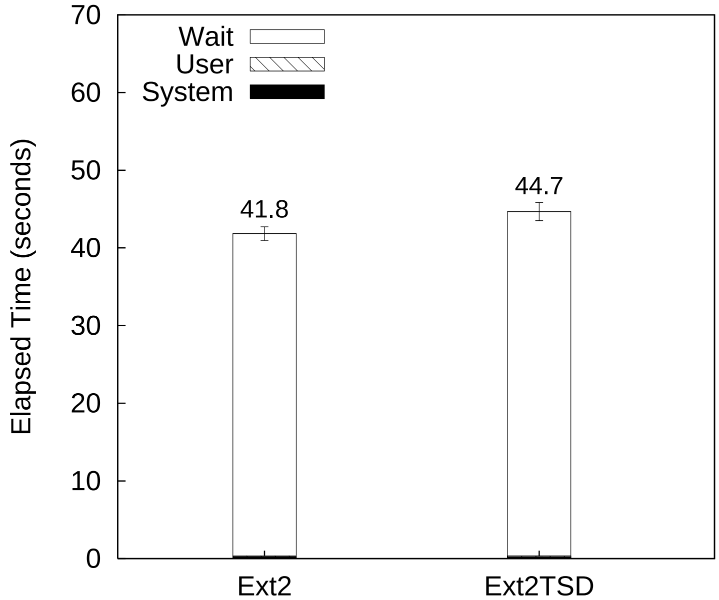


Figure 3.10: *Sprite LFS benchmarks: Random write results*

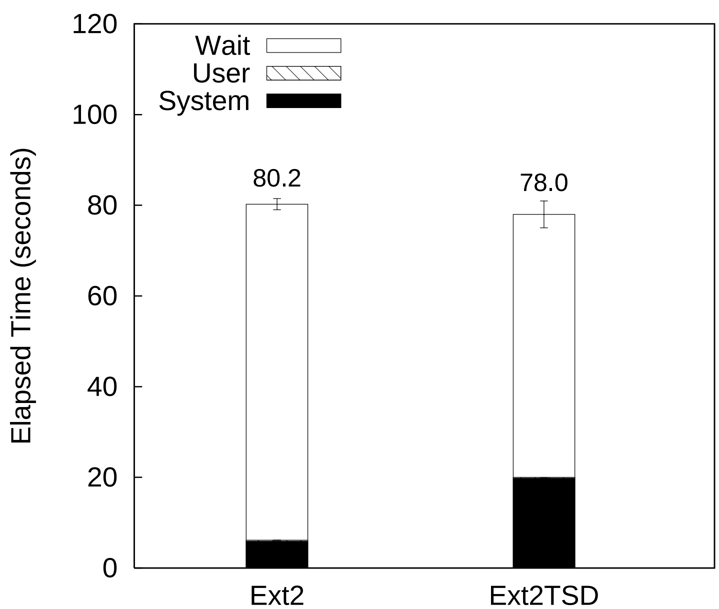


Figure 3.11: *Sprite LFS benchmarks: Sequential read results*

between Ext2TSD and regular Ext2 was negligible. However, the system overhead in Ext2TSD was 2.3 times. This was offset by a 21% reduction in wait time. As this is a sequential workload, a very large number I/O operations were executed within a short time interval. This resulted in making CPU overheads more visible. The CPU overheads were due to lock contention for the request queue shared by the main benchmark thread and the DPROTO disk thread. Our implementation uses a `spin_lock` for this, and hence it shows up as system time. The wait time decrease is because of better spatial locality in the case of Ext2TSD.

Figure 3.12 shows the results for sequential writes. The overheads of Ext2TSD were similar to sequential reads, as our sequential write workload performed overwrites of existing file data, resulting no additional pointer operations.

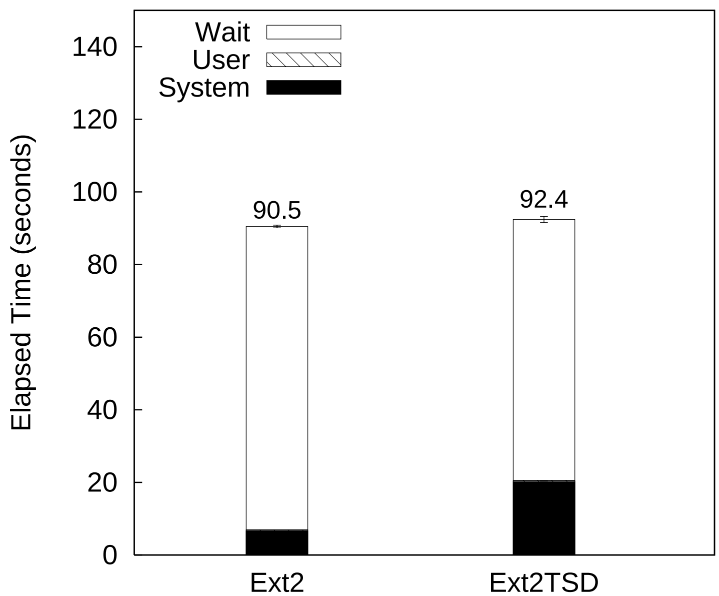


Figure 3.12: *Sprite LFS benchmarks: Sequential write results*

In summary, our evaluation shows that the overheads associated with our TSD disk infrastructure and the Ext2TSD file system is quite minimal (about 2%) for normal user workloads. This is shown by the results of our kernel compilation benchmark. For more I/O-intensive workloads such as Postmark and Sprite meta-data benchmarks, Ext2TSD shows overheads as high as 23%. We used such benchmarks to show the worst case overheads of TSDs. However, such I/O-intensive workloads are uncommon in real scenarios. Most of the system-time overheads were caused by pointer operations issued by the file system. This could be reduced by aggregating the operations and sending it to the disk system in batches. While the allocation primitive has to be synchronous, pointer creation and deletion can be made asynchronous.

Chapter 4

Case Study: ACCESS

We describe how type-safety can enable a disk to provide better security properties than existing storage systems. We designed and implemented a secure storage system called ACCESS (***A*** *Capability* ***C****onscious* ***E****xtended* ***S****torage* ***S****ystem*) using the TSD framework; we then built a file system on top, called Ext2ACCESS. We first motivate the need for enforcing disk-level capabilities, then present a detailed design of ACCESS. Finally, we describe our prototype implementation of ACCESS and the implementation of Ext2ACCESS, a file system that supports ACCESS.

Protecting data confidentiality and integrity during intrusions is crucial: attackers should not be able to read or write on-disk data even if they gain root privileges. One solution is to use encryption [12, 89]; this ensures that intruders cannot decipher the data they steal. However, encryption does not protect the data from being overwritten or destroyed. An alternative is to use explicit disk-level *capabilities* to control access to data [2, 25]. By enforcing capabilities independently, a disk enables an additional perimeter of security even if the OS is compromised. Others explored using disk-level versioning that never overwrites blocks, thus enabling the recovery of pre-attack data [74].

ACCESS is a type-safe disk that uses pointer information to enforce *implicit path-based* capabilities, obviating the need to maintain explicit capabilities for all blocks, yet providing similar guarantees.

ACCESS has five design goals. (1) Provide an infrastructure to limit the scope of confidentiality breaches on data stored on local disks even when the attacker has root privileges or the OS and file systems are compromised. (2) The infrastructure should also enable protection of stored data against damage even in the event of a network intruder gaining access to the raw disk interface. (3) Support efficient and easy revocation of authentication keys, which should not require costly re-encryptions upon revocation. (4) Enable applications to use the infrastructure to build strong and easy-to-use security features. (5) Support data recovery through administrative interfaces even when authentication tokens are lost.

4.1 Design

The primitive unit of storage in today's commodity disks is a fixed-size disk block. Authenticating every block access using a capability is too costly in terms of performance and usability. Therefore, there needs to be some criteria by which blocks are grouped and authenticated together. Since TSDs can differentiate between normal data and pointers, they can perform logical grouping of blocks based on the reference blocks pointing to them. For example, in Ext2 all data blocks pointed to by the same indirect block belong to the same file.

ACCESS provides the following guarantee: a block x cannot be accessed unless a valid reference block y that points to this block x is accessed. This guarantee implies that protecting access to data simply translates to protecting access to the reference blocks. Such grouping is also consistent with the fact that users often arrange files of related importance into individual folders. Therefore, in ACCESS, a single capability would be sufficient to protect a logical working set of user files. Reducing the number of capabilities required is not only more efficient, but also more convenient for users.

In ACCESS, blocks can have two capability strings: a `read` and a `write` capability (we call these *explicit capabilities*). Blocks with associated explicit capabilities, which we call *protected* blocks, can be read or written only by providing the appropriate capability. By performing an operation on a block Ref using a valid capability, the user gets an *implicit capability* to perform the same operation on all blocks pointed to by Ref, which are not directly protected (capability inheritance). If a particular reference block i points to another block j with associated explicit capabilities, then the implicit capability of i is not sufficient to access j; the explicit capability of j is needed to perform operations on it.

As all data and reference blocks are accessed using valid pointers stored on disk, root blocks are used to bootstrap the operations. In ACCESS, there are three kinds of access modes: (1) All protected blocks are accessed by providing the appropriate capability for the operation. (2) Blocks which are not protected can inherit their capability from an authenticated parent block. (3) Root blocks can be accessed without any reference block by providing the appropriate capability, if they are protected.

4.1.1 ACCESS meta-data

ACCESS maintains a table named KTABLE indexed by the block number, to store the blocks' `read` and `write` capabilities. During every block access it checks if the block has a KTABLE entry. If there is a KTABLE entry, the capability provided by the user is authenticated against the stored capability before performing the operation. ACCESS tracks the list of all reference blocks that are accessed successfully in a given period of time, and uses it to authenticate accesses to the blocks that do not have associated capabilities.

ACCESS also maintains a temporal access table called LTABLE which is indexed by the reference block number. The LTABLE has entries for all reference blocks whose associated implicit capabilities have not timed out. The timed out entries in the LTABLE are periodically purged.

4.1.2 Preventing replay attacks

In ACCESS, data needs to be protected even in situations where the OS is compromised. Passing clear-text capabilities through the OS interface could lead to replay attacks by a silent intruder who eavesdrops capabilities. To protect against this, ACCESS associates a sequence number with capability tokens. To read a protected block, the user has to provide a HMAC checksum of the capability (C_u) concatenated with a sequence number (S_u) ($H_u = HMAC(C_u + S_u, C_u)$). This can be generated using an external key card or a hand-held device that shares sequence numbers with the ACCESS disk system. Each user has one of these external devices, and ACCESS tracks sequence numbers for each user's external device. Upon receiving H_u for a block, ACCESS retrieves the capability token for that block from the KTABLE and computes $H_A = HMAC(C_A + S_A, C_A)$, where C_A and S_A are the capability and sequence number for the block, and are maintained by ACCESS. If H_u and H_A do not match, ACCESS denies access. Skews in sequence numbers are handled by allowing a window of valid sequence numbers at any given time.

4.1.3 ACCESS operation

During every reference block access, an optional timeout interval ($Interval$) can be provided, during which the implicit capabilities associated with that reference block will be active. Whenever a reference block Ref is accessed successfully, an LTABLE entry is added for it. This entry stays until $Interval$ expires. It is during this period of time, that we call the *temporal window*, all child blocks of Ref which are not protected inherit the implicit capability of accessing Ref. Once the timeout interval expires, all further accesses to the child blocks are denied. This condition should be captured by the upper level software, which should prompt the user for the capability token, and then call the disk primitive to renew the timeout interval for Ref. The value of $Interval$ can be set based on the security and convenience requirements. Long-running applications that are not interactive in nature should choose larger timeout intervals.

At any instant of time when the OS is compromised, the subset of blocks whose temporal window is active will be vulnerable to attack. This subset would be a small fraction of the entire disk data. The amount of data vulnerable during OS compromises can be reduced by choosing short timeout intervals. One can also force the timeout of the temporal window using the FORCE_TIMEOUT disk primitive described below.

To read a data block in ACCESS, the base pointer should be read first from one of the root blocks, by presenting the appropriate capability. If the access of the root block is successful, ACCESS will add an entry for the root block in the LTABLE. Once this is done, all blocks pointed to by the root block that do not have associated capabilities can be accessed until the LTABLE entry times out. In the context of a file system, the initial root block read would be its super block, and this occurs during `mount`. The temporal locality of the initial super block access is used as an implicit capability for accessing subsequent blocks. Whenever an implicit capability for a block needs to be verified, the disk checks if the reference block passed by the upper level software has an LTABLE entry for it. If an entry does not exist, ACCESS denies access to the block. If the reference block has an LTABLE entry, ACCESS looks up the PTABLE to find if the reference block indeed has a pointer to the block whose implicit capability needs to be verified. The reference block passed by the upper level software is only used for optimizing performance during the temporal lookup.

For blocks with associated capabilities, the appropriate capability string must be provided. Each reference block can have its own read and write capabilities depending on the owner of that reference block. For example, an indirect block of a particular user's file will have that user's capabilities, and cannot be read by anyone other than that person.

4.1.4 ACCESS API

To design the ACCESS API, we extended the TSD API (Section 3.2) with capabilities, and added new primitives for managing capabilities and timeouts. Note that some of the primitives described below let the file system specify the reference block through which the implicit capability chain is established. However, as we describe later, this is only used as a hint by the disk system for performance reasons; ACCESS maintains its own structures that validate whether the specified reference block was indeed accessed, and it has a pointer to the actual block being accessed. In this section when we refer to read or write *capabilities*, we mean the HMAC of the corresponding capabilities and a sequence number.

1. SET_CAPLEN$(Length)$: Sets the length of capability tokens. This setting is global.

2. ALLOC_BLOCKS$(Ref, Ref_r or C_w, Count)$: Operates similar to the TSD ALLOC_BLOCKS primitive with the following two changes. (1) If Ref is protected the call takes the write capability of Ref, C_w; (2) otherwise, the call takes the reference block Ref_r of Ref, to verify that the caller has write access to Ref.

3. ALLOC_CONTIG_BLOCKS$(Ref, Ref_r or C_w, Count)$: Same as the ALLOC_BLOCKS primitive, but allocates contiguous blocks.

4. READ$(Bno, Ref or C_{rw}, Timeout)$: Reads the block represented by Bno. Ref is the reference block that has a pointer to Bno. C_{rw} is either the read or the write capability of block Bno. The second argument of this primitive must be Ref if Bno is not protected for read, and must be C_{rw} if Bno is protected. $Timeout$ is the timeout interval.

5. WRITE$(Bno, Ref or C_w, timeout)$: Writes the block represented by Bno. C_w is the write capability of Bno. Other semantics are similar to READ.

6. CREATE_PTR$(Src, Dest, Ref_s or C_{sw}, C_{dw} or Ref_{dw})$: Creates a pointer from block Src to block $Dest$. If Src or $Dest$ are protected, their capabilities have to be provided. For blocks which are not protected, the caller must provide valid reference blocks which point to Src and $Dest$. Note that although the pointer is created only from the source block, we need the write capability for the destination block as well; without this requirement, one can create a pointer to any arbitrary block and gain implicit write capabilities on that block.

7. DELETE_PTR$(Src, Dest, Ref_s or C_{sw})$: Deletes a pointer from block Src to block $Dest$. Write credentials for Src has to be provided.

8. KEY_CONTROL$(Bno, C_{ow}, C_{nr}, C_{nw}, Ref)$: This sets, unsets, or changes the read and write capabilities associated with the block Bno. C_{ow} is the old write capability

of Bno. C_{nr} and C_{nw} are the new read and write capabilities respectively. A reference block Ref that has a pointer to Bno needs to be passed only while setting the write key for a block that did not have a write capability before. For all other operations, like unsetting keys or changing keys, Ref need not be specified because C_{ow} can be used for authentication.

9. RENEW_CAPABILITY$(Ref, C_{rw}, Interval)$: Renews the capability for a given reference block. C_{rw} is the read or write key associated with Ref. $Interval$ is the timeout interval for the renewal.

10. FORCE_TIMEOUT(Ref): Times out the implicit capabilities associated with reference block Ref.

11. SET_BLOCKSIZE and GET_FREE TSD primitives (Section 3.2) can be called through the secure administrative interface discussed in Section 4.3.

4.2 Path-Based Capabilities

Capability systems often use capabilities at the granularity of *objects* (e.g., physical disk blocks, or memory pages); each object is associated with a capability that needs to be presented to gain access.

In contrast, the implicit capabilities used by ACCESS are *path-level*. In other words, they authenticate an access based on the path through which the access was made. This mechanism of authenticating paths instead of individual objects is quite powerful in enabling applications to encode arbitrary trust relationships in those paths. For example, a database system could have a policy of allowing any user to access a specific row in a table by doing an index lookup of a 64-bit key, but restrict scans of the entire table only to privileged users. With per-block (or per-row) capabilities, this policy cannot be enforced at the disk unless the disk is aware of the scan and index lookup operations. With path-based capabilities, the database system could simply encode this policy by constructing two separate pointer chains: one going from each block in the table to the next, and another from the index block to the corresponding table block—and just have different keys for the start of both these chains. Thus, the same on-disk data item can be differentiated for different *application-level* operations, while the disk is oblivious to these operations.

Another benefit of the path-based capability abstraction is that it enables richer modes of sharing in a file system context. Let's assume there are n users in a file system and each user shares a subset of files with another user. With traditional encryption or per-object capability systems, users has to use a separate key for each other user that shares their files; this is clearly a key management nightmare (with arbitrary sharing, we would need n^2 keys). In our model, users can use the same key regardless of how many users share pieces of their data. To enable another user to share a file, all that needs to be done is a separate link be created from the other user's directory to this specific file. The link operation needs to take capabilities of both users, but once the operation is complete, the very fact that the pointer linkage exists will enable the sharing, but at the same time limit the sharing to only those pieces of data explicitly shared.

4.3 Key Revocation and Data Recovery

ACCESS enables efficient and easy key revocation. In normal encryption based security systems, key revocation could become pretty costly in proportion to the size of the data, as all data have to be decrypted and re-encrypted with the new key. With ACCESS, one just changes the capability for the reference blocks instead of the entire set of data blocks. Data need not be modified at all while revoking capabilities. This is one of the main advantages of ACCESS compared to traditional encryption-based security systems.

Secure key backup is a major task in any encryption-based data protection system. Once an encryption key is lost, usually the data is fully lost and cannot be recovered. This is ironical because a mechanism used for protecting data results in making the data inaccessible. ACCESS does not have this major problem. Data is not encrypted at all, and hence even if keys are lost, data can be retrieved or the keys may be reset using the administrative interface described below.

Often system administrators need to perform backup and or administrative operations for which the restricted ACCESS interface might not be sufficient. ACCESS will have a secure administrative interface, which could be through a special hardware port requiring physical access, in combination with a master key. Using the secure administrative interface, the administrator can backup files, delete unimportant files, etc., because the data is not stored internally in encrypted format.

4.4 ACCESS Prototype

We extended our TSD prototype to implement ACCESS. We implemented additional hash tables for storing the KTABLE and LTABLE required for tracking capabilities and temporal access locality respectively. All in-memory hash tables were periodically committed to disk through an asynchronous commit thread. The allocation and pointer management `ioctls` in TSD were modified to take capabilities or reference blocks as additional arguments. We implemented the KEY_CONTROL primitive as a new `ioctl` in our pseudo-device driver.

To authenticate the `read` and `write` operations, we implemented a new `ioctl`, KEY_INPUT. We did this to simplify our implementation and not modify the generic block driver. The KEY_INPUT `ioctl` takes the block number and the capabilities (or reference blocks) as arguments. The upper level software should call this `ioctl` before every read or write operation to authenticate the access. Internally, the disk validates the credentials provided during the `ioctl` and stores the success or failure state of the authentication. When a read or write request is received, ACCESS checks the state of the previous KEY_INPUT for the particular block to allow or disallow access. Once access is allowed for an operation, the success state is reset. When a valid KEY_INPUT is not followed by a subsequent read or write for the block (e.g., due to software bugs), we time out the success state after a certain time interval. This method of using an `ioctl` for sending the credentials greatly simplified our prototype implementation, as we did not have to modify the generic block driver interfaces to send additional arguments during the read and write operations.

4.5 The Ext2ACCESS File System

We modified the Ext2TSD file system described in Section 3.3.1 to support ACCESS; we call the new file system *Ext2ACCESS*. To demonstrate a usage model of ACCESS disks, we protected only the inode blocks of Ext2ACCESS with read and write capabilities. All other data blocks and indirect blocks had implicit capabilities inherited from their inode blocks. This way users can have a single read or write capability for accessing a whole file. An alternative approach may be to protect only directory inode blocks. ACCESS provides an infrastructure for implementing security at different levels, which upper level software can use as needed.

To implement per-file capabilities, we modified the Ext2 inode allocation algorithm. Ext2 stores several inodes in a single block; so in Ext2ACCESS we needed to ensure that an inode block has only those inodes that share the same capabilities. To handle this, we associated a *capability table* with every isegment (Section 3.3.1). The capability table persistently stores the checksums of the capabilities of every inode block in the particular isegment. Whenever a new inode needs to be allocated, an isegment is chosen using the inode allocation algorithm of Ext2, and then the isegment is scanned for an inode block with a matching capability for the new inode to be created. If a matching block is found, the inode is allocated in that block, otherwise a free block is chosen from the isegment. When a new block is chosen, the capability checksum for that block is updated in the capability table. If there are no free blocks left in the isegment, the algorithm searches forward in the remaining isegments.

Ext2ACCESS has two file system `ioctl`s, called SET_KEY and UNSET_KEY, which can be used by user processes to set and unset capabilities for files. These `ioctl`s take the pathname as an argument. When users need to create a new protected file, they have to call the `set_key ioctl` before the create. Ext2ACCESS then associates the capability with the newly created pathname, and then performs inode allocation appropriately. For subsequent operations on the file, the user has to provide the right capability before the operations. The life of a user's key in kernel memory can be decided by the user. For example, a user can call the SET_KEY `ioctl` before an operation and then immediately call the UNSET_KEY `ioctl` after the operation is completed to erase the capability from kernel memory; in this case the life of the key in kernel memory is limited to a single operation. Ext2ACCESS uses the KEY_INPUT device `ioctl` of ACCESS to send the user's key before reading an inode block. For all other blocks, it sends the corresponding reference block as an implicit capability, for temporal authentication.

An issue that arises in Ext2ACCESS is that general file system meta-data such as super block and descriptors need to be written to all the time (and hence must have their capabilities in memory). This can potentially make them vulnerable to modifications by attackers. We address this vulnerability by mapping these blocks to root blocks and enforce that no pointer creations or deletions can be made to root blocks except through an administrative interface. Accordingly, `mkfs` creates set of pointers to the relevant inode bitmap and isegment descriptor blocks, but this cannot change after that. Thus, we ensure confidentiality and write protection of all protected user files and directories.

Although the above solution protects user data during attacks, the contents of the meta-data blocks themselves could be modified (for example, free block count, inode allocation status, etc). Although most of this information can be reconstructed by querying the pointer structure from the disk, certain pieces of information are hard to reconstruct. Our current implementation does not handle this scenario, but there are various solutions to this problem. First, we could impose that the disk perform periodic snapshotting of root blocks; since these are very few in number, the overhead of snapshotting will be minimal. This enables an administrator to recover those root blocks to a previous snapshot in the event of an attacker modifying these blocks. Alternatively, some amount of NVRAM could be used to buffer writes to these global metadata blocks and periodically (say once a day) an administrator "commits" these blocks to disk using a special capability after verifying its integrity.

4.6 Evaluation

We evaluated the performance of ACCESS using our Ext2ACCESS file system. We compared Ext2ACCESS with a regular Ext2 file system mounted on a regular disk. The hardware setup we used was same as that for evaluating the TSD infrastructure, described in Section 3.6. We ran three different workloads: Postmark, kernel compilation, and Sprite LFS meta-data benchmark. We discuss the results of our evaluation below:

4.6.1 Postmark Results

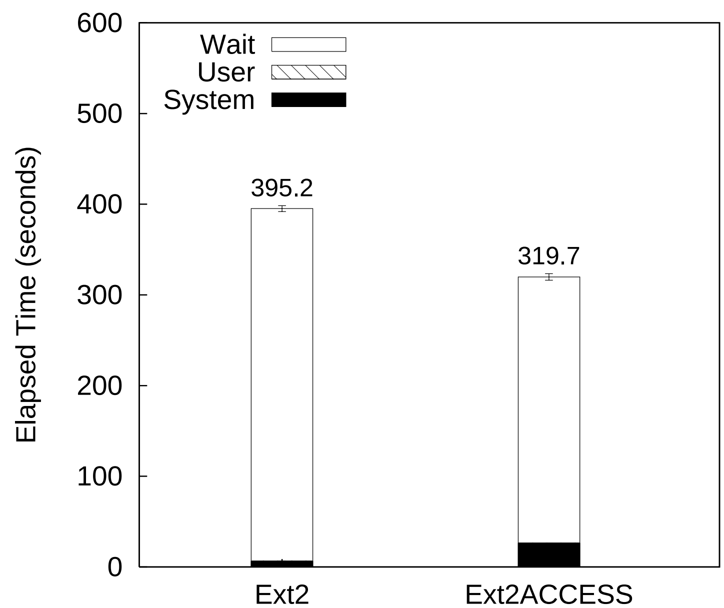


Figure 4.1: *Postmark Results for ACCESS*

Figure 4.1 shows the results for Postmark. For this benchmark, we configured Postmark with 10,000 files of sizes ranging from 100KB to 200KB, and 10,000 transactions. Ext2ACCESS performed 19% better than regular Ext2, mainly because of a 24% decrease in I/O time. The difference in I/O time in this case is more than that of Ext2TSD vs. regular Ext2 discussed in Section 3.6 because ACCESS pre-allocates more memory than regular TSD for its data-structures. This results in reduced cache size making the impact of spatial locality more pronounced. The system time for Ext2ACCESS was 3 times more than that of regular Ext2 mainly because of pointer and key management `ioctls` issued by Ext2ACCESS.

4.6.2 Kernel Compile Results

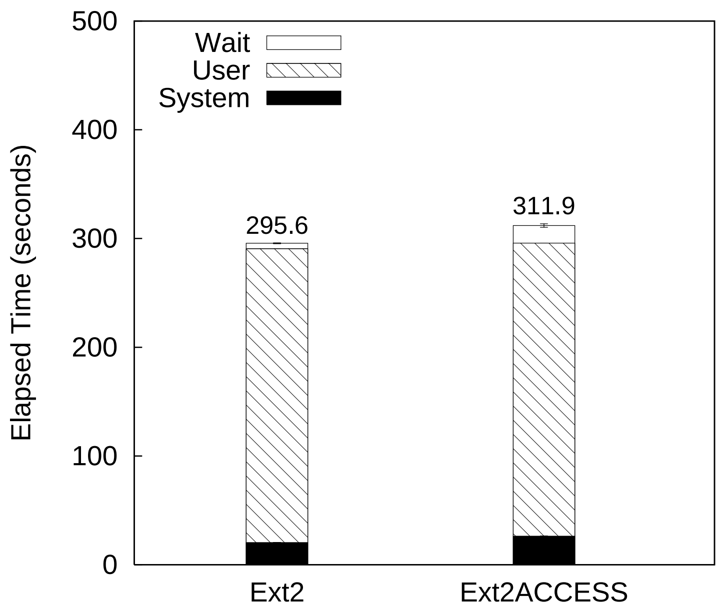


Figure 4.2: *Kernel Compile Results for ACCESS*

Figure 4.2 shows the kernel compilation results for Ext2ACCESS. As evident from the figure, the overall elapsed time overhead of Ext2ACCESS was 5% compared to regular Ext2. This is caused by a 29% increase in system time and 2.2 times increase in wait time. The wait time increase in this case is because the compilation thread waits for the disk thread to service the key management and pointer operations. The wait time is more pronounced in this benchmark compared to Postmark, because kernel compilation has a small I/O component by virtue of its CPU-intensive nature.

4.6.3 Sprite LFS Benchmark Results

We ran the Sprite LFS meta-data benchmarks consisting of file creating, reading, and file deletion. We used the same setup as described in Section 3.6.

Figure 4.3 shows the results for the file creation phase. Ext2ACCESS had an elapsed time overhead of 10%. The system time overhead was 110% mainly caused due to a large number of pointer and key management operations. As this is a meta-data I/O-intensive

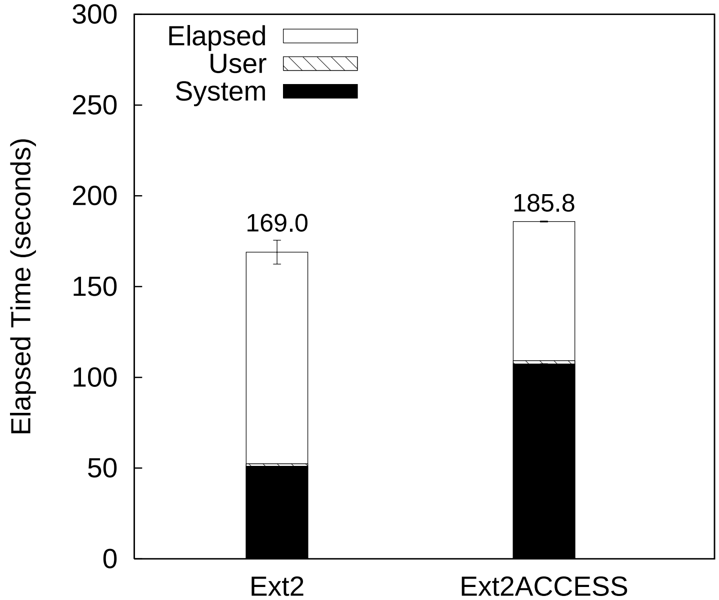


Figure 4.3: *Sprite LFS benchmark: Create results for ACCESS*

workload, the system time increase is more pronounced. The wait time for Ext2ACCESS reduced by 34% because of better spatial locality.

As shown in Figure 4.4, the elapsed time overheads for Ext2ACCESS for the read phase was negligible. However, there is a 22% increase in system time caused by key management `ioctls`. The system time increase is smaller compared to the create phase, as this is a read-only benchmark and hence pointer operations do not occur.

Figure 4.5 shows the overheads of Ext2ACCESS for the delete phase of the Sprite LFS meta-data benchmarks. The elapsed time overhead was 19% compared to regular Ext2. This is because of a 149% increase in system time. The system time increase is because of a large number of pointer deletion operations within a short time interval, as file `unlink` operations results in a smaller amount of I/O.

Overall, ACCESS has more system time overheads compared to regular TSD. This is mainly because of additional key management `ioctls` such as KEY_INPUT that Ext2ACCESS had to call while reading meta-data. For normal user workloads (such as kernel compilation), ACCESS has a small overhead of 5%.

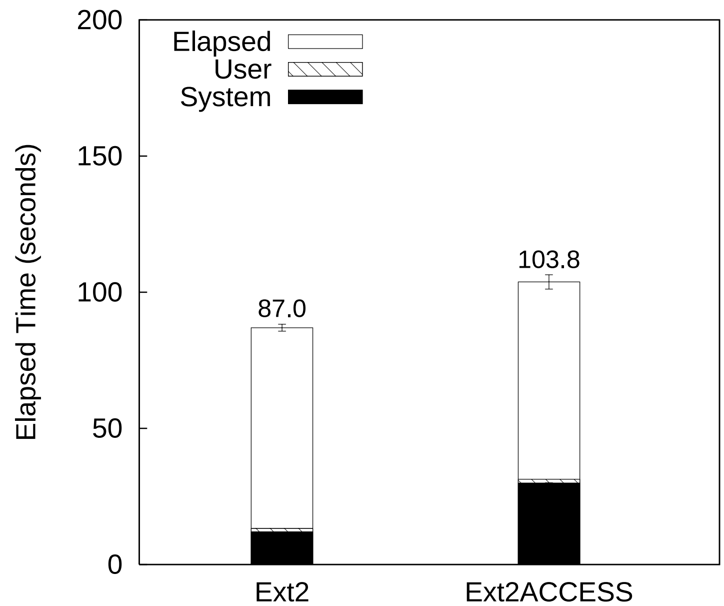


Figure 4.4: *Sprite LFS benchmark: Read results for ACCESS*

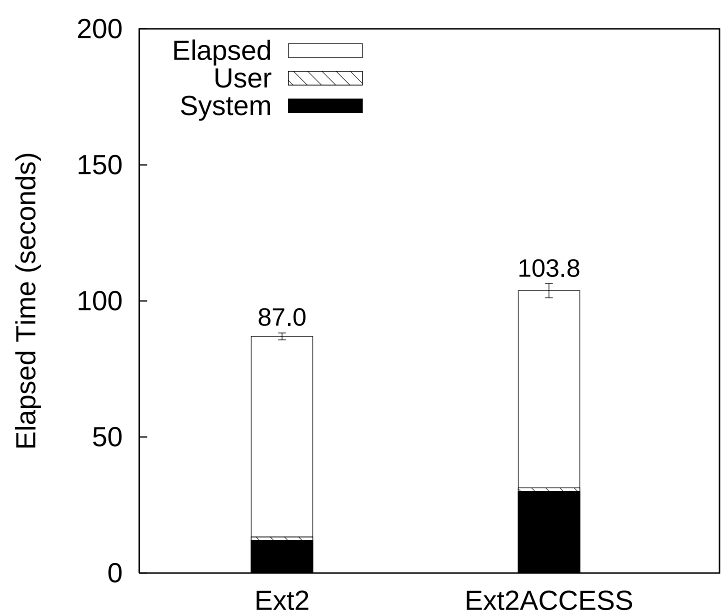


Figure 4.5: *Sprite LFS benchmark: Delete results for ACCESS*

Chapter 5

Case Study: Disk-level Data Consistency

A key challenge in persistent data storage on disk is ensuring the *consistency* of data in the face of crashes. In many cases, on-disk data is unusable unless it conforms to certain software-specific invariants that define its consistency. For example, an on-disk B-Tree with dangling pointers in some of its nodes cannot be used to locate data items. Similarly, in a file system, a directory pointing to invalid or unallocated inodes constitutes a consistency violation.

Given the importance of consistency, most file systems and other software that manage on-disk storage incorporate mechanisms to ensure on-disk consistency. While some techniques involve optimistically updating on-disk state and then *fixing* consistency violations based on a disk scan (e.g., `fsck`), more modern techniques such as journalling [27] or Soft updates [24] involve constraining updates in such a way that consistency is enforced. These mechanisms are quite complex; for example, modern file systems owe a significant portion of their complexity to satisfying this requirement.

This traditional approach to managing consistency entirely at the file system or software is fraught with two key weaknesses. First, the disk system is completely oblivious to the consistency of the data it stores, which constrains the range of functionality it can provide. For example, today's block-based disk systems cannot perform consistent snapshotting of data. Snapshotting is a popular and useful feature in the storage industry, but consistent snapshotting has so far been restricted only to storage systems exporting a richer NFS-like interface [32]. Similarly, modern storage systems perform backup and asynchronous remote mirroring [39]; consistency-awareness at the storage level can increase the utility of these techniques.

A second problem with the current approach to consistency management is that every file system and every software layer that manages on-disk data is forced to duplicate the mechanisms needed to enforce consistency. This raises the bar for implementing any disk-resident data structures. Although applications can use generic transactional libraries, it often requires restructuring the application to be aware of transactions and tracking transaction context across concurrent, asynchronous operations. For example, although the journalling block device (JBD) layer in Ext3 provides a transactional interface, the Ext3 codebase had to go through a substantial amount of restructuring to actually use JBD [81].

To address these problems, we present *ACE-Disk*, an **A**utomatic **C**onsistency **E**nforcing Disk, a disk system that preserves the semantic consistency of stored data. In our approach, the disk system takes responsibility for consistency management, and thus is empowered to provide consistency-aware functionality such as snapshotting. Applications simply inform the disk about the relationship between various blocks that the application already knows about. Specifically, we advocate using a *Type-Safe Disk* (TSD) [68], a disk system that is aware of the pointer relationship between blocks, to get consistency, with minimal modifications at the software-level.

Our disk-level consistency mechanism enforces the following constraint: the on-disk version of data should always be consistent. To accomplish this, we need to discover semantically consistent groups of blocks and commit them atomically to the disk when they are written by higher level software such as the file system. All inconsistent block updates should be buffered inside the disk until they become consistent. For example, when a new file is created, the corresponding directory block and the inode block have to be updated. When just one of the writes arrives at the disk it indicates an inconsistent update. In that case, we need to buffer the update until the second block write also arrives. When both the directory block and inode block writes have arrived at the disk, we need to ensure (at the disk level) that both these blocks are committed atomically to stable storage.

In this section, we describe the main aspects of our disk-level consistency mechanism. First, we discuss some related work. Second, we describe how update dependencies between blocks can be inferred from pointers. Third, we present our enhanced pointer interface that make dependency inference robust. Fourth, we describe the consistency enforcement process a key issue in disk-level consistency enforcement. We finally detail our prototype implementation of the system, and discuss some limitations of pointer-driven consistency.

5.1 Inferring Dependencies from Pointers

Determining semantic relationships between blocks at the disk level requires additional information exchange between the software layer and the disk. Today's block-based disks treat all stored information as opaque data and they do not have knowledge of data semantics. For example, today's disks cannot differentiate between a data and meta-data block in a file system. We leverage the idea of Type-Safe Disks (TSDs) [68], to obtain pointer-relationships between blocks as maintained by the higher level software.

Pointers at the disk level not only convey structural information about data items stored on disk, but also they enable the disk to infer dynamic relationships between blocks that get updated. For example, when a new block a is allocated and a pointer is created to it from another block b, both a and b depend on each other. If the system crashes when just one of the blocks is updated, the disk is left in an inconsistent state. This is because, if only block a is updated, it would be pointing to a block with junk data (not yet written), and if only b is updated, it becomes unreachable as there would be no incoming pointers to it.

The existing TSD interface consists of primitives for allocation and pointer operations as discussed in Section 3.2. We discuss how each TSD primitive can be used to infer update dependencies.

The allocation primitive internally creates a pointer to the newly allocated block, in the reference block passed. This operation relates two blocks: the newly allocated block and the reference block. Updating one of the blocks alone clearly leaves the system in an inconsistent state; hence these two blocks constitute a dependency constraint and they have to be committed atomically to stable storage.

The pointer creation primitive creates a pointer from any two arbitrary allocated blocks. In this case, the source block *must* be written subsequent to the pointer creation operation to write the new pointer value in it. However, the destination block need not necessarily be written, as the it is a previously allocated block. For example, while creating a new file in the Ext2 file system, a pointer gets created from the directory block to an already allocated inode block that contains the inode of the new file. In this case, both these blocks constitute a dependency. This is because the directory block has to be updated with the new pointer to the inode block, and the inode block has to be updated with valid information about the newly created file. Failure to commit the latter will result in a directory entry pointing to an invalid inode. As a counter example, if we consider a common index-based storage structure, a set of index blocks point to data block. In this case, duplicating an index block for reliability reasons would result in creation of new pointers from the duplicated index block to the existing data blocks. Here only the index block needs to be written and not the data blocks. Therefore, the pointer creation primitive provided by TSD does not convey enough information to decide whether or not the source and destination blocks constitute a dependency.

A pointer deletion operation deletes an existing pointer from block a to block b. This operation has a special case: if the deleted pointer is the last incoming pointer to block b, we garbage collect b and it can be re-allocated during future allocation requests. In both cases, it is clear that block a has to be written subsequent to this operation for it to reflect the pointer deletion. The destination block b in the case of garbage collection need not be written. However, it does constitute a dependency: b must not be re-allocated until a is written. For example, when the last pointer from an inode block to a data block is deleted during a `truncate` operation, re-allocating the data block to another inode before the old inode is written could result in a state where the old inode points to the contents of a different file. In the normal case of a pointer deletion where garbage collection does not occur, we cannot infer whether the source and destination constitute a dependency for the same reason as explained in the case of pointer creation.

5.2 An Enhanced Pointer Interface

As described in the previous section, the pointer API exported by a TSD do not always convey enough information to make correct inferences in a generic manner. In this work, we fine-tune the TSD API to make it more complete in terms of conveying pointer information.

We introduce the notion of a *sub-block* in a TSD. We use sub-blocks to formalize *allocatable* units inside a block, as maintained by the higher-level software. For example, in Ext2 each inode block can contain several inodes, each of them allocated and freed at the software level. Although formalizing these units in a precise manner requires knowledge about the unit size and offsets inside a block, we just need a rudimentary knowledge of sub-blocks to infer dependencies. For example, to decide whether or not a create or delete pointer operation constitutes a dependency we just need to know if that pointer points to a sub-block. This intuition is based on the fact that, to preserve pointer consistency we need to guarantee two properties: first, no pointer points to unwritten (junk) units, and second, no allocated units become unreachable. In our inference mechanism we make use of additional disk primitives for creating and deleting pointers to sub-blocks. Note that the disk need not track information about sub-blocks, but it just needs to dynamically know sub-block pointer operations by way of explicit primitives. Higher-level software call the respective sub-block primitives while creating and deleting pointers to newly allocated or freed sub-blocks. For example, Ext2 has to call a sub-block pointer creation primitive to create a pointer between a directory block and inode block while creating a file. From this we can infer that the directory and inode blocks form a dependency constraint.

We present an extended pointer interface to TSDs that captures most cases of dependency inferences. In the primitives described below, the parameter t refers to a logical timestamp value for the operation. This is to let the disk know about the temporal ordering of operations as they are issued by the higher level software. The purpose and usage of this parameter is discussed in detail later in this section.

1. READ($Blockno$): Block read primitive.
2. WRITE($Blockno$, t): Block write primitive.
3. ALLOC_BLOCK(Ref, t): Allocates a new block a from the disk-maintained free-block list and creates a pointer to it in Ref. Both Ref and a constitute a write dependency constraint.
4. CREATE_PTR(Src, $Dest$, t): Creates a new pointer from Src to $Dest$. This primitive does not create any dependency.
5. DELETE_PTR(Src, $Dest$, t): Deletes an existing pointer from Src to $Dest$. If this is the last incoming pointer to $Dest$, $Dest$ is garbage collected (marked free) and it creates a new dependency between the write of Src and the re-allocation of $Dest$.
6. MOVE_PTR(Src, $Dest$, $Newsrc$, t): Moves the source block of an existing pointer from Src to $Newsrc$. This operation results in creation of a new dependency for the writes of Src and $Newsrc$. This primitive is useful for handle cases such as a `rename` operation in a file system, or a B-tree node split where pointers need to be moved from one block to another.
7. ALLOC_SUB_BLOCK(Ref, $Target$, t): Creates a new pointer between block Ref and block $Target$. $Target$ is a block that contains multiple allocatable software-level structures. This primitive is called when a software-level structure in $Target$ is allocated. This disk does not track these structures. This creates a new write dependency between Ref and $Target$. The disk differentiates this primitive from the CREATE_PTR primitive only to infer dependencies.
8. FREE_SUB_BLOCK_PTR(Ref, $Target$, t): Deletes an existing pointer between Ref and $Target$. $Target$ is a block that contains multiple allocatable software-level

structures. This primitive is called when a software-level structure in $Target$ is freed. If this operation deletes the last incoming pointer to block $Target$, $Target$ is garbage collected and a new dependency is created between Ref update and re-allocation of $Target$. If the pointer deleted is not the last incoming pointer to $Target$, a new dependency is created for the update of Ref and $Target$.

5.3 Consistency Enforcement

In this Section we detail how an ACE-disk guarantees consistent data commits to stable storage. Figure 5.1 shows the overall architecture of an ACE-disk.

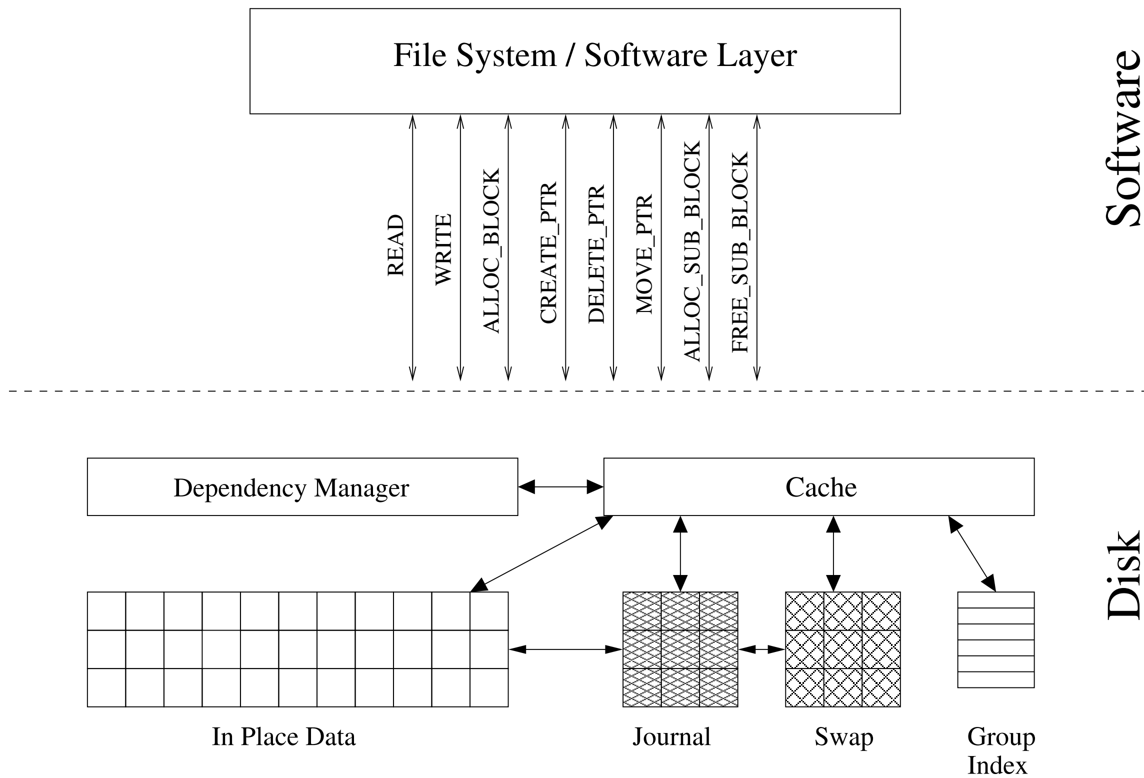


Figure 5.1: *Architecture of an ACE-disk*

An ACE-disk consists of five main components: (1) *dependency buffer*, a buffer layer made of high-speed memory where inconsistent block updates are buffered until the corresponding dependency becomes consistent; (2)*buffer swap space*, a swap area in the disk which is used to swap out inconsistent buffer data when the cache is full; (3) *journal space*, an area on disk which is used to ensure atomic update of resolved dependencies; (4) *group manager*, which tracks the pointer operations and constructs dependencies; *group index* a data-structure used by the group manager to store disjoint dependencies and the blocks affected by each of those dependencies. The buffer layer acts both as a read and write cache, and gets invalidated during power down of the disk. All inconsistent block updates are buffered in the cache to ensure that the state of data stored in place is always consistent.

The swap space is used when the number of inconsistent blocks exceed the size of the high speed buffer memory.

When an ACE-disk infers a dependency during a pointer operation, it associates a *group* object with that dependency. This group object contains information about the set of blocks that are affected by that dependency. We use the terms *group object* and *dependency group* interchangeably in the rest of the report to refer to a list of blocks that needs to be committed atomically to stable storage to ensure consistency. A *group entry* refers to a member of a group which contains a block number and the time at which it was added. When a block is written *after* it is added to a dependency group, the corresponding group entry for that block is marked "ready." When all entries in a dependency group are ready, the group is said to be *resolved*, and all blocks associated with it can be committed atomically to the disk.

In a simple case, when the first pointer operation happens in a disk causing a dependency creation between two blocks a and b, a new dependency group G is created and both the blocks are added to it. When write requests for both a and b have arrived at the disk, the dependency group G is said to be *resolved* and all the blocks in G can be committed atomically to the disk. However, if another pointer operation happens before G is resolved introducing a dependency between blocks b and c, the operation *extends* the existing dependency group. This is because, one of the blocks in the new dependency (block b) is already part of an existing dependency. Thus, in this scenario block c should be added to group G as well. Therefore, whenever there is a new dependency introduced between any two blocks x and y by way of a pointer operation, one of the following three actions are taken:

1. If both x and y are not part of any existing dependencies, a new dependency group is created and x and y are added to it.

2. If only one of x or y is associated with an existing dependency group G, then both blocks are associated with G and are marked "not ready."

3. If both x and y are already associated with the same group G, then no group action needs to be taken. However, the entries in the group pertaining to blocks x and y have to be marked "not ready" as a new constraint is added between the two blocks.

4. If both x and y are associated with different groups G_1 and G_2, then G_1 and G_2 are *merged*, and the entries for x and y are marked "not ready"

As pointer operations construct dependencies between blocks, higher-level software must ensure that the pointer management primitives are issued to the disk *before* the source and destination blocks are updated. This constraint is implicitly enforced for the block allocation primitive as a block cannot be updated before it is allocated. However for the pointer creation and deletion primitives, higher-level software has to ensure that it follows this ordering rule. For example, when a `create` happens in Ext2, the sub-block pointer creation primitive has to be issued for the directory and the inode blocks before the contents of the blocks are updated.

5.3.1 Temporal Ordering of Operations

ACE-disk's consistency mechanism relies on the temporal relationships between operations seen at the disk level. For example, an entry in a dependency group is marked ready when a write arrives after the dependency creation. However, in today's modern operating systems and disks, operations can be re-ordered at any level. For example, file systems today predominantly perform asynchronous I/O where block writes are buffered at the software level and are flushed to the disk in regular intervals of time. Moreover, modern disk device drivers re-order or merge disk requests before issuing to the disk for performance reasons. These factors make the temporal ordering of operations that the disk sees completely different from the order that the higher-level software issued. Therefore, unless additional ordering information is communicated from the software-level, the disk cannot obtain the precise temporal order of operations.

ACE-disk solves this problem by introducing two constraints on the operations: (a) all pointer primitives take place synchronously and (b) all operations have associated logical timestamps. These two constraints enable the disk to obtain precise temporal ordering of the operations. Although synchronous pointer operations may affect performance, it is mitigated by the fact that these operations do not result in block I/O inside the disk, in the critical path. Timestamps in this case are logical. For example they can be a monotonically increasing sequence number. Whenever higher-level software issues a pointer operation, it has to pass a sequence number along with it. Similarly when the in-memory copy of a disk block is updated by the software, a sequence number has to be associated with the buffer for that block. Whenever a pointer operation introduces a dependency, its sequence number is associated with the corresponding group entries. The entries are marked ready only when a subsequent write arrives with sequence number greater than the stored one. Note that introducing sequence numbers with block I/O operations is simple—we have modified the Linux kernel to support sequence numbers along with buffers whenever they are dirtied. This modification was trivial and required changing just 50 lines of code.

When a dependency group is resolved all blocks in the group has to be committed in place atomically. A power failure while committing a dependency group should not leave the in place data in an inconsistent state. ACE-disk uses a logging mechanism to ensure this. All blocks in a resolved groups are first written to a log and synced with a commit identifier before the in place commit happens. The log is discarded when the in place commit is complete. After a crash, an ACE-disk checks the log for valid group data and replays them. The log contains separate journals for each dependency group and hence each of them are replayed after the crash to bring the system to a consistent state.

5.4 Bounding Commit Interval

The amount of data lost during a crash depends on the interval between the instant a block write arrives at the disk and the time when it is actually committed to stable storage. In an ACE-disk, inconsistent block data gets buffered until the entire dependency group is resolved. ACE-disk's mechanism of managing dependency groups allow extending a group whenever pointer operations happen from or to a member of the group. Thus, during normal operation, a dependency group could potentially get extended repeatedly during a continuous workload that performs pointer operations. For example, in Ext2, for a recursive directory creation workload, the entire working set would form part of the same dependency group as all blocks branch out from the inode of the root directory. Moreover, as pointer operations always precede the block write operations, a dependency group could never get resolved for a continuous workload. This is because before the time when all blocks in a group are marked ready, the group could be extended several times with new blocks or new dependencies for the existing blocks. This results in two problems. First, large amounts of data may get lost in the event of a crash, although the on-disk state is consistent. Second, excessively long dependency groups require buffering of a large number of blocks and hence impose onerous space requirements.

Bounding the interval between dependency commits is challenging particularly at the disk level because the disk has no knowledge about intermediate versions of block data that are known to the higher-level software. This is because most higher-level software buffer writes and hence the versions of block data that reach the disk could be a small subset of total number of versions that the software knows about. For example, if a file is created in Ext2, an inode block is modified. Before the inode block write is issued to the disk, if another file is created whose inode is in the same block, the disk sees only the version of the block updated with both inodes. Therefore, the disk cannot spawn a new dependency group during a pointer operation for a block, when the existing group containing a block has reached a time threshold.

Blocking pointer operations at the disk level until an existing dependency is committed could be a solution to the bounding problem, but requires radical modifications to the higher-level software to support it. This is because software such as file systems perform locking of data-structures at an operation level. When a pointer operation blocks, the file system could sleep after grabbing a lock on the data-structure which reside on a block that needs to be committed for some dependency to resolve. This could result in a deadlock as the block containing the data-structure cannot be committed until the operation in execution completes.

An ACE disk solves this problem by having new error modes for pointer creation operations. The allocation and pointer management primitives could optionally return one of the following errors to the higher-level software: SYNC_BOTH, SYNC_SRC, or SYNC_DEST. As the names indicate, the disk can fail a pointer operation and choose to request the higher level software to write the source, destination, or both blocks associated with that operation. Upon receiving one of these errors the software should issue a write of the current version of the corresponding blocks, and then retry the pointer operation. At the disk level, whenever a dependency group is unresolved beyond a time threshold it is *frozen*. Whenever new dependencies are created for a block that is already part of a frozen group and in an “not ready” state, the disk returns one of three errors mentioned above, depending on whether the block is the source, destination, or when both the source and destination blocks exist in frozen groups in “not ready” state. This way of forcing the software to commit the intermediate version of the data helps the disk to spawn new dependency groups for blocks that are already ready in a frozen group. An ACE-disk ensures that at a block is never part of more than two groups at a time, the older of which is frozen. This is done by ensuring that a group is not frozen until all blocks in the group are not part of any other frozen group. This method ensures commit of dependency groups in tune with the block write interval of the higher level software. We verified the correctness of our bounding solution by implementing this in the Ext2 file system. Each every case, the commit interval of the dependency groups were in tune with that of the software level write-back interval.

5.5 Implementation

We implemented a prototype ACE-disk as a pseudo-device driver in the Linux kernel 2.6.15 that stacks on top of an existing disk block driver. The pseudo device driver layer receives all block requests, and redirects the common read and write requests to the lower level device driver after the required processing. The additional primitives required for operations such as block allocation and pointer management are implemented as driver `ioctls`.

To enable sequence numbers with block I/O requests, we added a new field to the buffer header object and the `request` token object in the Linux kernel. Whenever a buffer is marked dirty, we generate a sequence number and update it in the buffer header. When a write is issued for a buffer, the sequence number is carried over to the `request` object and hence available to the ACE-disk pseudo-device driver. Sequence numbers are generated by an atomic increment of a counter value. The same counter value is used during pointer operations and modifying buffers. Our prototype ACE-disk contained 6900 lines of kernel code of which 3060 lines of code were reused from the existing TSD prototype.

5.6 Limitations of Pointer-driven Consistency

While the update dependency information conveyed by pointers is quite rich and as we show, sufficient to enforce consistency, it has some limitations when compared to the more general notion of transactional consistency. Specifically, the dependency information conveyed by pointers is limited to a pair of blocks; e.g. if a pointer is created between two blocks, the two blocks will be updated atomically. However, our mechanism cannot support atomic commits of an arbitrary group of blocks. For example, on creation of a new directory (i.e. mkdir) in ext2, a pointer is created from the parent directory block to the inode of the child directory, and the inode initialized. Then a new block is allocated for the child directory and a pointer created between the child inode and the child directory's new data block. With a transactional system, these three blocks will be committed atomically. But in our case, the first pointer creation and the initialized inode could be committed before the second pointer creation. As a result, a directory inode may end up with a state where it has no blocks at all, which is an apparent violation of consistency.

However, we argue that this consistency problem falls under a class of *online-patchable* consistency violations. For example, just by looking at the initialized directory inode with no pointers, it is unambiguous that a crash happened just before the new directory's block got allocated, so it's safe to immediately allocate a new block for the directory and assign it to the inode. Note that in contrast, a more "real" consistency problem would be a directory pointing to the wrong inode, perhaps a regular file inode, where it is not obvious what the correct state should be. Pointer consistency could lead to such transient online-patchable consistency violations the violation is readily and unambiguously identifiable and the fix for that is obvious as well. Most importantly, the fix to such a violation is *local*, in that it does not require looking at the global state of the file system. We believe that the pointer-derived consistency semantics is thus a useful and simpler counterpart to the more general transactional consistency.

5.7 Evaluation

We evaluated the performance of our prototype ACE-disk using Ext2ACE. We ran both a general purpose workload and a micro-benchmarks on our implementation and compared it with a regular Ext2 and Ext3 file systems running on a normal disk. We compared our system with Ext3 because it is a journalling file system that provides similar consistency guarantees as ACE-disk at the software level. For all benchmarks we used Ext3 in its default journalling mode (ordered writes mode). In this mode file meta-data alone is journalled and it is written to the journal only after the corresponding data blocks are written directly in place.

For all benchmarks we included the file system unmount time in our calculation. This is because ACE-disk commits dependency groups asynchronously using separate kernel threads, and a file system unmount procedure blocks until all outstanding threads have completed their commit operation. This is relevant even for normal Ext2 and Ext3 as they commit all outstanding dirty data during an unmount.

5.7.1 Postmark Results

We configured Postmark to create 30,000 files whose sizes ranging from 512 bytes to 10 KB, and perform 250,000 operations in 200 directories. This workload particularly stresses the ACE-disk as a large number of dependencies get created and resolved during the meta-data operations. The time taken for the Postmark benchmark for Ext2, Ext3, and Ext2ACE are shown in Figure 5.2.

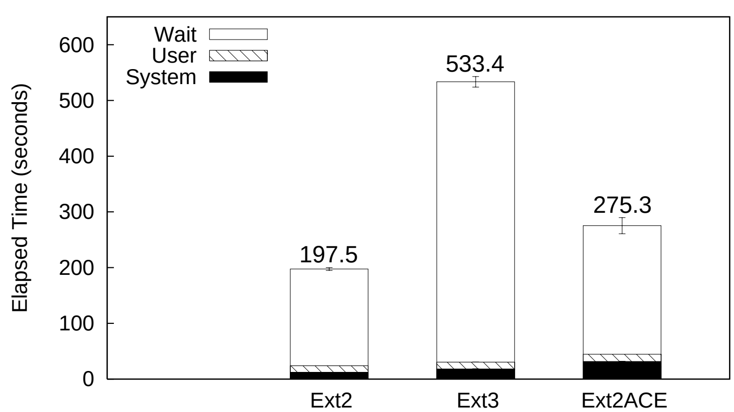


Figure 5.2: *Postmark Results for ACE-Disk*

Ext2ACE on top of ACE-disk had an elapsed time overhead of 40% compared to regular Ext2 on a normal disk. Although the system time increase is 2.6 times relatively, this has not contributed much to the elapsed time overhead. As mentioned earlier, this overhead is because of dependency tracking during every block write and pointer operations. The wait time increase (32%) is predominantly because all blocks are written out twice in the case of an ACE-disk to ensure atomic commits of dependency groups. All block data is written out to the journal first and after the journal is synced, in-place commits happen. Ext3 ran almost twice as slow as Ext2 because of its ordered journalling mode. Ext2ACE is faster than Ext3 in this case because ACE-disk journals both data and meta-data blocks and for a small file workload such as Postmark, random writes get converted to sequential ones. The in-place commit of data in ACE-disk happens in an asynchronous manner.

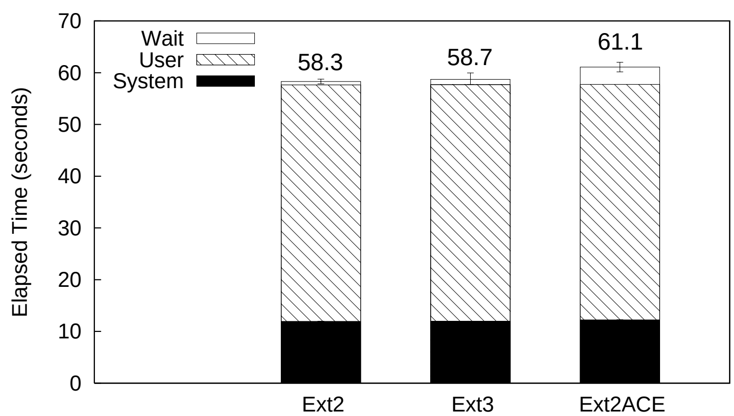


Figure 5.3: *Openssh Compile Results for ACE-Disk*

5.7.2 Compile Benchmark Results

To simulate a relatively CPU-intensive user workload, we compiled the OpenSSH source code. We used OpenSSH version 4.5, and analyzed the overheads of Ext3 and Ext2ACE for the `untar`, `configure`, and `make` stages combined. These operations in combination constitute a significant amount of CPU and I/O operations. The results for OpenSSH compilation is shown in Figure 5.3.

The times taken by Ext2 and Ext3 for the compilation workload are almost similar. This is because this is a mostly CPU-intensive workload. Ext2ACE had an elapsed time overhead of 5% compared to Ext2 and Ext3. This is because of the increase in wait time (1 sec vs. 3.4 secs). The increase in wait time is caused by the CPU context switches between the main compilation process and the asynchronous dependency commit threads of ACE-disk. Since this is a CPU-intensive workload, the context switch time is more pronounced than Postmark. In a real environment, as the dependency commits are performed inside the disk, this context switch overhead would not be seen. The system time overhead is not significant for Ext2ACE in this case because there are relatively few I/O operations that require processing to track dependencies.

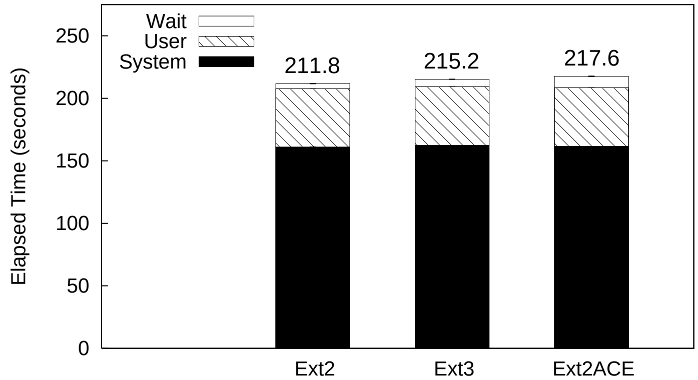


Figure 5.4: *Create Micro-Benchmark Results for ACE-Disk*

5.7.3 Micro-benchmarks

We ran two micro-benchmarks to obtain the overheads of the `create` and `unlink` file system operations. We evaluated these two operations because both of them exercise the ACE-disk's dependency trackers and consistency enforcement mechanism. For the create workload, we created 500 directories with 1,000 files each totaling to 500,000 files. For the unlink workload, we removed all created files and directories. The results of the `create` and `unlink` workloads are shown in Figures 5.4 and 5.5, respectively.

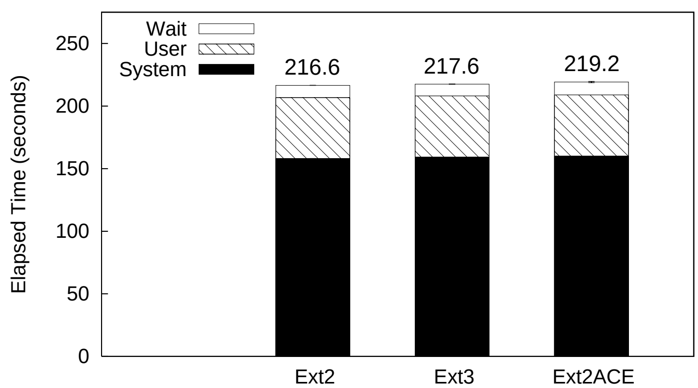


Figure 5.5: *Unlink Micro-Benchmark Results for ACE-Disk*

For the `create` workload, Ext2ACE had an overhead 2.7% compared to Ext2. This is mostly caused by the increase in wait time due to the additional I/O operations writing out block data twice for ensuring atomicity in block commits. For the `unlink` workload the results of Ext2ACE is similar to Ext2 and Ext3 as `unlink` results in smaller number of writes than creates, because freed blocks are not written to the disk.

Overall ACE-disks have small overheads for normal user workloads. When the workload is highly I/O-intensive, more information needs to be tracked by the disk to manage dependencies. This results in more CPU time which is mitigated by the fact that the disk uses its own isolated CPU in a real environment.

Chapter 6

Case Study: Discriminating Hierarchical Storage System

Modern large storage systems are virtually supercomputers; a typical high-end storage system from EMC [16] or NetApp [37] has hundreds of processors, tens of gigabytes of RAM and hundreds of disks. In tune with the increasing processing power available at the storage systems, their functional sophistication has also increased. Today, storage systems employ various forms of RAID for reliability and performance, use non-volatile RAM to absorb write latency, perform dynamic block migration for load balancing, and so on [16, 88].

Although storage systems have evolved significantly in terms of the range of functionality they provide, they are still constrained due to one fundamental limitation: they have little or no information about the system layers above that use the storage system, and thus view data simply as a flat stream of bytes. For example, they do not know what pieces of data are more important than others, what pieces are likely to be accessed randomly vs. sequentially, etc. Although a lot of storage-level policies such as RAID level, caching policy, etc. can be tuned for specific kinds of usage, a typical storage system cannot fully exploit this potential because it deals with a myriad of interleaved types of data each with different access characteristics, and has very little information to separate these types from each other.

In this section, we present DHIS (pronounced as *this*), a **D**iscriminating **Hi**erarchical **S**torage system, that uses various hints specified from the higher layers about the type of the data to select custom policies for managing the data, such as the exact RAID level, cacheability of the data in NVRAM etc. DHIS also uses information on the logical relationship between blocks conveyed in the form of logical pointers [68] to extrapolate its type information from one identifying block to its descendants. By being able to discriminate between data with varying requirements, DHIS is able to balance conflicting goals such as performance and reliability much more efficiently than traditional storage systems.

To make informed choices on the exact layout and caching policies to use for a specific piece of data, DHIS enables the layers above to annotate logical chunks of data with *attributes* on the data. For instance, the file system can specify that a given file (identified by the top-level inode block for the file) will be mostly subject to small random writes. Given this attribute associated with the file, DHIS would make sure to not place the file in a

RAID-5 format, given the "small-write" performance penalty incurred in RAID-5; instead, it may choose to place it in RAID-1 (mirroring) format.

There are five attributes that DHIS supports: importance of the data (which determines how reliably the data should be stored), the normal access-pattern on the data (i.e., random or sequential), the expected popularity of the data (i.e., hot or cold), whether the data is read-mostly or write-mostly, and finally, the expected lifetime of the data (i.e., whether it corresponds to a temporary file). Based on these five attributes, DHIS decides on the specific redundancy and reliability scheme to use for the data, and the various forms of caching to use (e.g., whether to cache the data in NVRAM or perhaps a faster Flash storage layer) such that the best performance/reliability trade-off is obtained. Specifically, the current implementation of DHIS utilizes these attributes to automatically select the RAID level a piece of data goes to, and to decide which pieces of data to cache in NVRAM.

We evaluate DHIS using our software-level disk prototyping framework. Using this we evaluate the various discriminating policies of DHIS and demonstrate their effectiveness. We show that DHIS can achieve significant performance wins by exploiting higher-level attributes. We show that the flexibility to choose RAID-levels on a per-file basis provides significant benefits in performance, compared to the one-size-fits-all solution normally employed in today's systems. We also show that by intelligent caching of data that is subject to frequent random writes (e.g., meta-data blocks in a file system) in NVRAM, DHIS greatly improves overall system performance.

The rest of this section is organized as follows: in the next sub-section, we discuss the background of modern storage systems and type-aware storage. In Section 6.1, we describe the design details of DHIS and show the kind of optimizations that DHIS enables. Section 6.4 presents our disk protototyping framework and our prototype implementation of DHIS. We evaluate our prototyping framework and our implementation of DHIS in Section 6.5.

6.1 Design

In this section, we describe the design of DHIS in detail and discuss the optimizations that DHIS achieves by using higher-level attributes on data. We first describe the type-aware hierarchical storage setup that DHIS incorporates and its salient features. We then present the set of well-defined higher-level attributes that DHIS supports, and finally we show the kind of optimizations that these attributes enable.

6.1.1 A Hierarchical Storage Architecture

DHIS's architecture comprises volatile and NVRAM, as well as several individual disks aggregated using standard RAID levels. In our design, we particularly consider the three most commonly used RAID levels: RAID0 (striping without redundancy), RAID1 (mirroring), and RAID5 (striping with a parity block per stripe). These three RAID levels have varying characteristics in terms of performance, reliability, and cost per gigabyte. We aim to use these resources within a single storage system and manage them efficiently in a transparent manner using higher-level hints about data and access semantics. DHIS exports a flat namespace to higher-level storage software such as file systems, and aggregates the storage capacity available in the different RAID levels internally. The architecture we use while designing DHIS is shown in Figure 6.1.

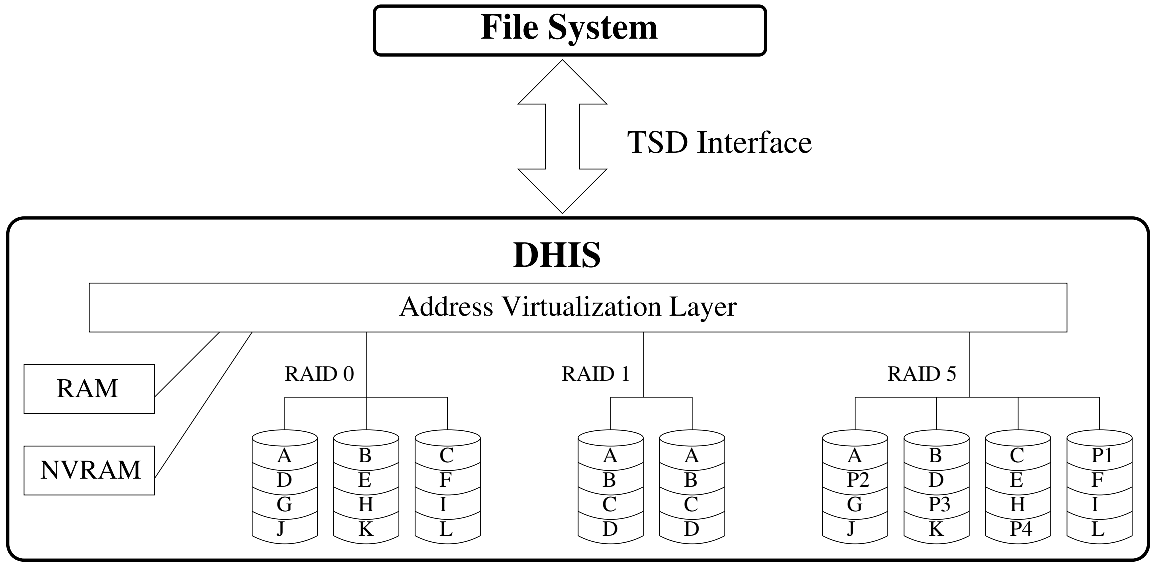


Figure 6.1: *DHIS Setup*

In the rest of this section, we detail the basic design aspects of operating such a hierarchical storage system in a type-aware storage setup.

Virtualizing the Block Layer Namespace Although DHIS manages several disks and RAID levels internally, it appears like a single disk system to higher-level software. For this purpose, it maintains a block-address virtualization layer that contains an address translation table, `TTABLE`, which maps the global logical block namespace to individual disk-specific addresses. A physical address contains two parts: a disk or device identifier (e.g., an internal RAID device), and a physical block number within that device. The `TTABLE` is looked up for every I/O request, and is updated whenever blocks need to be re-mapped to different devices. DHIS stores the `TTABLE` and other book-keeping structures in non-volatile RAM and periodically writes them to the disk. Note that inbuilt non-volatile memory has been quite common in high-end storage devices for a while, and recently it is being used even for regular hard drives [76].

Block Allocation DHIS performs free-space management at the firmware level, thereby freeing higher-level applications from maintaining information solely for placement of data on disk. Block allocation is done using an explicit `alloc_block` disk primitive. This is important for two reasons. First, higher-level software is unaware of internal disk characteristics and hence cannot make correct decisions about block locality especially when the storage system has a complex hierarchy of disk media internally. For example, an Ext2 file system's allocation algorithm assumes that blocks whose logical block numbers are contiguous are physically contiguous as well. This may not be true in a hierarchical storage system. Second, by managing free-space on disk, DHIS can exploit its knowledge of block-liveness to proactively perform operations such as aggressive replication of hot read-only data, to improve performance and reliability. The block allocation API optionally takes a hint block number to allocate the new block closer to it.

One of the main design goals of DHIS is to enable placement of data blocks at the right RAID-level based on higher level data characteristics such as access patterns, relative importance, etc. Therefore, whenever a block is allocated by the higher-level, the disk has to assign a logical block number for it in the global block namespace, and then allocate a physical block in one of the RAID devices. To enable this, DHIS maintains an allocation bitmap for the logical namespace and separate bitmaps for every underlying physical device. The block-allocation primitive performs two steps: one to allocate logical block number and the second for a physical block number in one of the lower disks. A `TTABLE` entry is added whenever a new block is allocated.

6.1.2 Pointer-Based Optimizations

Just like a TSD, DHIS tracks pointer information across blocks. DHIS includes disk primitives, `CREATE_PTR(srcblk, destblk)` and `DELETE_PTR(srcblk, destblk)` that higher-level software can use to communicate to DHIS whenever a logical pointer is created or deleted. DHIS maintains all pointers with respect to the global logical block namespace, and not the physical blocks. This allows DHIS to relocate physical blocks transparently without affecting the stored pointer information.

By using pointer knowledge, DHIS performs three key optimizations as follows:

1. All higher-level meta-data blocks (identified as those having outgoing pointers) are placed in the RAID level of highest reliability and best random-access performance. This is because meta-data blocks are more important and accessed more frequently compared to regular data blocks. In our setup, we use RAID1 for this purpose. Note that as the physical destination of blocks are determined at the time of allocation, we do not have information about outgoing pointers for a newly created block and hence we cannot differentiate between data and meta-data for a newly allocated block. Only when the first outgoing pointer is created from a block, DHIS can identify it as a meta-data block. Therefore, DHIS performs dynamic relocation of meta-data blocks to RAID1 as and when the first pointer is created from a block.

2. As meta-data blocks need to be written to disk frequently for reliability reasons, DHIS attempts to absorb the write latency of these blocks by caching writes in NVRAM. As meta-data blocks constitute a small percentage of the total size of storage, NVRAM caching is beneficial. DHIS flushes out the NVRAM contents to RAID1 in configurable periodic intervals of time and also when the device is idle.

3. DHIS exploits its knowledge about block-liveness (differentiating between used and unused blocks) to remove dead blocks (those freed by the file system) from the NVRAM cache and the regular disk cache, for improving the cache utilization.

6.2 Attributes

In this section we describe the set of hints or attributes that higher-level software such as file systems can associate with disk blocks in DHIS. Note that knowledge about pointers at the disk level allows DHIS to inherit attributes of a meta-data block to the sub-tree of data blocks that it points to. For example, to set an attribute for a file, a file system just needs to set an inheritable attribute to the per-file meta-data block, and DHIS automatically inherits the attribute to all blocks belonging to that file.

There have been previous efforts to infer the characteristics of blocks at the disk level without an explicit interface, by using history of accesses [88] or block correlations [46]. However, these methods are quite limited in the range of characteristics they can infer, and often end up being too complex. For example, although it is possible to identify hot and cold blocks using access history, information such as the relative importance of blocks with respect to higher-level applications cannot be inferred easily. Therefore, DHIS provides an explicit interface for communicating a set of well-defined hints or attributes that can be set by higher-level software such as file systems.

6.2.1 Attribute Interface

Higher-level software can set attributes using an explicit disk primitive, `DHIS_SETATTR`, by passing a bitmap representing the attributes. Note that attributes in DHIS are normally set to meta-data blocks, and they qualify the characteristics of all blocks in the pointer tree starting from that block. For example, if an Ext2 file system needs to specify the access pattern for a file, it needs to set an appropriate attribute to the corresponding inode block. DHIS automatically groups blocks in the sub-tree and associates the attribute to all such blocks. The following are the attributes that DHIS supports:

- `IMPORTANCE`: Determines the relative importance of a data item. Currently DHIS supports this as a boolean attribute which indicates that an entity is more important than others. This can potentially be extended to supports more fine-grained levels based on the diversity in internal storage hardware. Applications can set this attribute for source files or documents that need to be preserved with the highest level of reliability.

- `ACCESS_PATTERN`: Determines if the set of blocks (belonging to the sub-tree of a meta-data block) will be accessed at random or sequential. This attribute takes three values: not set, random, or sequential. Applications can set this attribute to files they own based on their access pattern. For example, a simple classification of files based on their types can enable a file system to mark video files as sequential and database index files as random.

- `HOT/COLD`: Specifies the frequency in which the particular data item will be accessed. This takes either of these three values: not set, hot, and cold. Generally applications can set archival data as cold and frequently updated files such as database write-ahead log files as hot.

- `READ-MOST/WRITE-MOST`: Indicates whether a data item will be mostly read or written. For example, binary files such as `/bin/ls` in Unix will be mostly read and will be updated only infrequently. Similarly, file system journals or database log files will predominantly be written.

- `TEMPORARY`: This is a boolean attribute that indicates whether a data item is temporary (i.e., short-lived) in nature. For example, object files generated by compilers and intermediate files generated by applications such as download managers can be classified as temporary.

Storage software such as file systems can set attributes for appropriate meta-data blocks, using application-specific information. For example, file systems can export an interface to user applications to set attributes at the granularity of files or directories. In such cases, file systems have the responsibility to transform logical abstractions (such as files) into corresponding meta-data blocks and to pass the attributes to DHIS. For example, an Ext2 file system can export an `ioctl` that user applications can use to set attributes to a file identified by a path name. Ext2 can then issue a `DHIS_SETATTR` call with the attribute, for the inode block corresponding to the path name.

6.2.2 The Ext2DHIS File System

We have developed an attributes-aware file system to support DHIS, as an extended form of the Ext2TSD file system [68]. Ext2TSD is a modified Ext2 file system that supports TSD devices. There are two main differences between a regular Ext2 file system and Ext2TSD. First, Ext2TSD does not perform free-space management, and allocates blocks using the TSD disk API. Second, whenever a new pointer is added or removed for a meta-data block (such as an inode), Ext2TSD issues the corresponding `CREATE_PTR` or `DELETE_PTR` calls to the disk to communicate the pointer.

We have developed Ext2DHIS as an extended Ext2TSD file system that includes an ioctl interface for user applications to set attributes to files or directories. Ext2DHIS issues `DHIS_SETATTR` calls to the storage system whenever attributes need to set or changed. In addition to this, we have developed a simple scheme to set basic attributes automatically for known file name extensions, at the file system level. For example, Ext2DHIS automatically marks files with extensions `.c`, `.cpp`, etc., as important as these may be source files. This provides a simple means to set basic attributes without the need to modify user-level applications.

6.3 Attribute-Based Optimizations

In this section, we describe the optimizations that DHIS achieves using the well-defined set of attributes listed above. First, we present the method we use to choose the right RAID level for a given data item. Second, we describe how better NVRAM utilization can be done by choosing the right candidates to cache. Third, we detail how information about temporary files can aid in reducing disk fragmentation.

6.3.1 Choosing Optimal RAID Level

The three RAID levels that DHIS manages have different performance and reliability characteristics. In this section, we first describe the key characteristics of RAID levels in DHIS and then we detail the policies DHIS adopts to choose the right RAID level to place data.

Characteristics of RAID Levels RAID0 performs plain striping across several disks without any redundancy and hence it has the lowest reliability level among the three. However, in terms of performance, RAID0 is good for sequential and random read-write workloads. This is mainly because I/O operations get parallelized across the individual disks when data is striped. In terms of cost per gigabyte, RAID0 is the cheapest as there is no redundancy and the storage capacity is the sum of the individual disk capacities.

RAID1 mirrors data across two disks. As two disks contain identical data at all times, data reliability is better as it can tolerate a single disk failure. In terms of performance, RAID1 has similar characteristics for both sequential and random I/O. Reads are faster than writes as reads can be parallized across the two disks. Write speed is in tune with that of a single disk, because for every write, both disks have to be updated, but in parallel. RAID1 has the highest cost per gigabyte as the total capacity of the drives is halved due to mirroring.

RAID5 stripes both data and parity information across three or more drives. In principle it is similar to having a single dedicated parity drive, but parity blocks are distributed across all drives RAID5 can recover from single disk failures and hence has comparable reliability to RAID1. Read performance in RAID5 is similar to that of RAID0. However, for small random writes RAID5 performs poorly. This is because for small writes that do not span a complete stripe, computation of new parity involves reading the old contents of the data block and the parity block. In terms of cost per gigabyte, RAID5 is the second best among the three, as there is a single parity block for a stripe.

RAID Placement Policies In addition to placing all meta-data blocks in RAID1 (as described in Section 6.1.2), DHIS also adopts placement policies based on higher-level attributes. Table 6.1 shows the placement policies that DHIS adopts for each combination of attributes. The principles that we use to decide the RAID level for a data item are in tune with the performance and reliability characteristics associated with each RAID level as described above. Note that for data that is `IMPORTANT` and `COLD` we use RAID5 irrespective of its access pattern and read-write characteristics because they are going to be accessed rarely and hence performance is not a significant factor.

6.3.2 Choosing Candidates for NVRAM caching

DHIS chooses candidates for NVRAM caching to maximize the number of absorbed writes through NVRAM. It chooses all meta-data blocks as candidates as described in Section 6.1.2, because meta-data blocks are frequently written and have random access patterns. Similarly, it also chooses blocks with the combination of attributes `HOT`, `WRITE-MOST`, and `RANDOM`, as these are expected to benefit the most from NVRAM

IMPORTANT	ACCESS_PATTERN	READ/WRITE-MOST	HOT/COLD	RAID Levels
No	Any	Any	Any	0, 5, 1
Yes	Any	Any	Cold	5, 1, 0
Yes	Not set	Not set	Not set or Hot	5, 1, 0
Yes	Random	Not-set or Write-most	Not set or Hot	1, 5, 0
Yes	Random	Read-most	Not set or Hot	5, 1, 0
Yes	Sequential	Any	Not set or Hot	5, 1, 0

Table 6.1: *RAID placement heuristics. The order of RAID levels listed in the last column is the desired order for each combination of attributes. DHIS tries the next level when allocation fails in one of the levels.*

caching. We do not choose sequential workloads as candidates and in general they do not benefit much from caching.

DHIS manages NVRAM buffers using a simple mechanism that caches writes when the corresponding block is a candidate, and a asynchronous process that flushes NVRAM buffers to disk whenever the disk is idle. When all buffers in the NVRAM are dirty, DHIS passes all subsequent writes to other candidates directly to disk, until NVRAM buffers are flushed out.

6.3.3 Reducing Disk Fragmentation

A fragmented disk can yield poor performance for large files that are accessed sequentially. This is because, when there are only fragments of free-space left for allocation, large files may end up spread out across the disk resulting in unnecessary disk seeks. Temporary files that get created and deleted within short intervals of time could exacerbate disk fragmentation thereby seriously affecting the performance of large files under some scenarios.

DHIS deals with temporary files in a different manner reducing disk fragmentation. As DHIS is responsible for free-space management, it allocates space for block groups with the TEMPORARY attribute set, in a segregated portion (group of blocks at the end of the device) of RAID level 0. For blocks that are not temporary, DHIS never allocates space from this segregated area. This ensures that temporary files that get created and deleted never interfers with the allocation of regular files, thereby significantly reducing disk fragmentation.

6.4 Prototype Implementation

We implemented a protoype of DHIS using our disk prototyping framework. We preallocated the size of each of our data-structures, TTABLE, NVRAM cache, allocation bitmaps, attribute and pointer management structures, and request queue, as a function of the total storage capacity. For the three RAID levels, we stacked DPROTO on top of the regular Linux software RAID drivers for RAID0, RAID1, and RAID5. Our prototype of DHIS had 2,150 lines of kernel code in addition to DPROTO.

6.5 Evaluation

We evaluated the performance of our prototype implementation of DHIS to get an estimate of the benefits achieved by attribute-based RAID placement and NVRAM caching. We first present our evaluation setup and describe the benchmarks that we used. We then show evaluation results for DHIS's RAID placement and NVRAM caching mechanisms.

6.5.1 Evaluation Setup

For all benchmarks, we used a 2.8GHz Xeon with 1GB RAM, and 6 250GB, LSILogic SCSI disks. We used Fedora Core 6, running a vanilla 2.6.15 kernel.

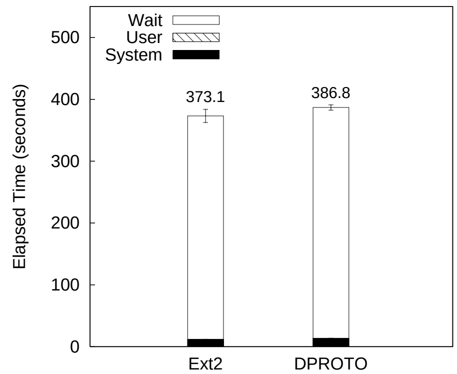
To ensure a cold cache between benchmark runs, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean. Wait time is the elapsed time less CPU time used and consists mostly of I/O, but process scheduling can also affect it.

We observed disk statistics from `/proc/diskstats` for each of our benchmarks and used it to verify the reasons behind our results. Disk statistics provide the following information observed by the disk for each benchmark we ran: number of read I/O requests (`rio`), number of write I/O requests (`wio`), number of sectors read (`rsect`), number of sectors written (`wsect`), number of read requests merged (`rmerge`), number of write requests merged (`wmerge`), total time taken for read requests (`ruse`), and the total time taken for write requests (`wuse`).

6.5.2 Benchmarks and Configurations

We used Postmark [82], a popular file system benchmarking tool, to test the performance of our prototypes. Postmark is I/O-intensive and stresses the file system by creating a large number of small files and then performing a series of file system operations such as directory lookups, creations, and deletions on them. A large number of small files is common in electronic mail and news servers where multiple users are randomly modifying small files. Postmark mostly generates a combination of small random reads and writes, and hence we use this for testing performance of our implementations, under random workloads. The working set of a Postmark benchmark is determined by the number of files to be created initially, and their size range. For all runs of Postmark we used file sizes ranging from 400KB to 600KB. We have mentioned the exact configuration of Postmark used for each test, along with the respective test results.

We also ran a series of micro-benchmarks to test the characteristics that Postmark does not cover. For example, Postmark does not evaluate sequential I/O performance and overheads for large file workloads. Micro-benchmarks also isolate the overheads for specific operations, and hence gives a clearer picture of the overheads. We developed a user-level tool that performs generates one of the following workloads: random read, random write, sequential read, and sequential write. For all runs, we used 4KB read or writes on a single 1.5GB file. For the sequential benchmarks (read and write), we performed sequential



r

Figure 6.2: *Postmark results for DPROTO vs. regular disk*

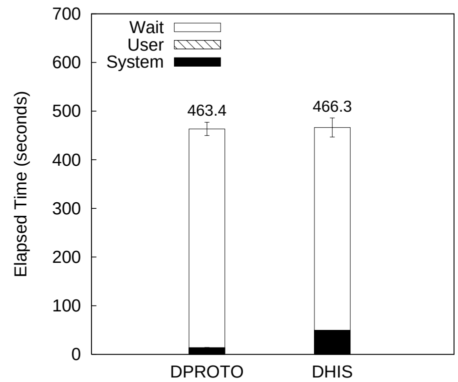


Figure 6.3: *Postmark results for Ext2DHIS over DHIS compared to Ext2 over plain DPROTO*

4K I/O on the 1.5GB file 5 times totalling to 7.5GB of I/O. For random read and write benchmarks, we performed 20,000 and 150,000 4K I/O respectively.

6.5.3 DHIS Results

We evaluated the performance of our prototype implementation of DHIS and our optimizations for RAID placement and NVRAM caching.

Figure 6.3 shows the overheads of DHIS over regular DPROTO. We configured DPROTO to preallocate the same amount of memory that DHIS required for storing its data-structures (128MB). Although the elapsed times for both runs are similar, DHIS has higher system time (13 secs vs. 49 secs) and lower wait time (449 secs vs. 416 secs) compared to regular DPROTO. The system time increase is because of two reasons. First, Ext2DHIS issues ioctls to the pseudo-device driver to communicate pointer information, contributing the major component of system time. Second, the shared queue is protected by a spin lock and hence minor contention causes a busy wait resulting in increased system time. The reduced wait time is because of better spatial locality caused by the disk-level block allocation scheme used by DHIS (compared to file-system–level allocation in Ext2)

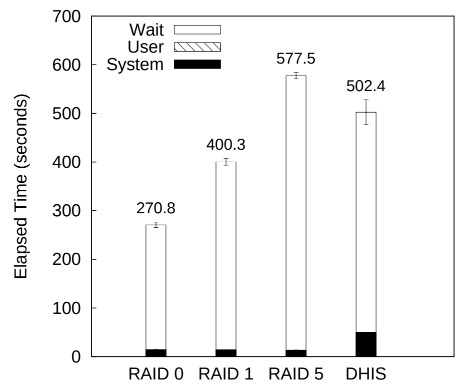


Figure 6.4: *Postmark results for Ext2DHIS over DHIS compared to Ext2 over plain DPROTO on RAID1 and RAID5*

which co-locates blocks in a greedy fashion without taking into account future file growth.

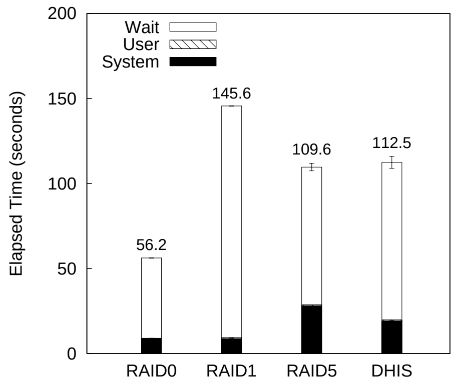
6.5.4 RAID Placement Optimizations

To evaluate the benefits of the RAID placement optimizations performed by DHIS, we used Postmark and micro-benchmarks. For all benchmarks, we observed the time taken for the workload on regular DPROTO stacked over individual RAID1 and RAID5 devices and compared them with DHIS. While running the workload over DHIS we set the `IMPORTANT` and `ACCESS_PATTERN` attributes set, so that DHIS would place them in the optimal RAID level.

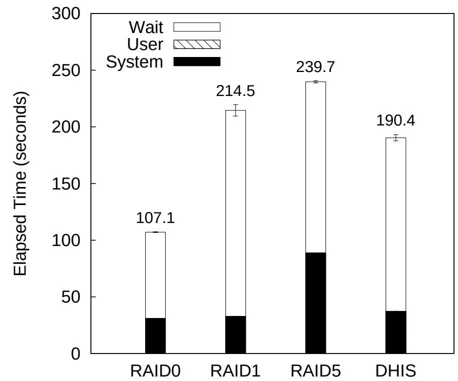
As Postmark generates mostly a random workload, we ran it with `RANDOM` attribute set. For micro-benchmarks, we set the `SEQUENTIAL` and `RANDOM` attributes respectively for sequential and random reads and writes.

Figure 6.4 shows the Postmark results for DHIS compared to DPROTO on individual RAID1 and RAID5. As evident from the figure, DHIS performs closer to regular RAID1 as it placed the Postmark working set on its RAID1 hierarchy. DHIS has an elapsed time overhead of 25% compared to regular RAID1 although DHIS places all data on RAID1, for this benchmark. This is because of two reasons. First, Postmark is creates and deletes a large number of files and hence results in a large amount of pointer operations and attribute updates. This results in increased system time (13 secs vs. 49 secs) as seen from the figure. Second, as pointer operations are synchronous in nature, they block until the DPROTO service thread handles them. This results in increased wait time (386 secs vs. 452 secs). The overheads are more pronounced for the Postmark workload because Postmark is an extreme case of I/O-intensive workload. In most common workloads DHIS performs much more closer to the RAID1 for random workloads (as shown in the micro-benchmark results below).

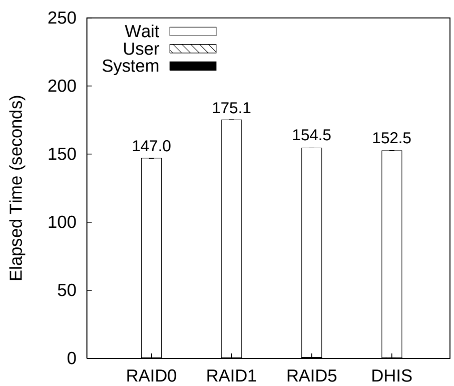
Figure 6.5 shows the micro-benchmark results for RAID placement. As shown in the graphs, under all cases, DHIS performs close to the fastest of the two RAID levels. Note that for the sequential write workload, DHIS performs 16% better than RAID5. This is because DHIS places all meta-data blocks in RAID1 for maximizing reliability and better



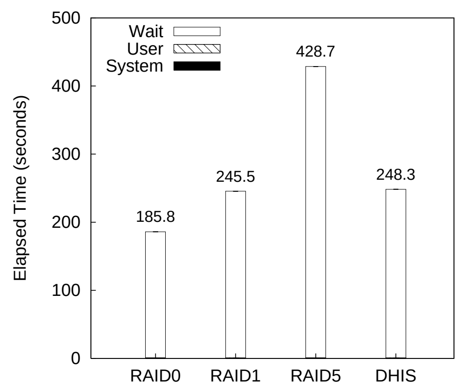
(a) Sequential Read Benchmark



(b) Sequential Write Benchmark



(c) Random Read Benchmark

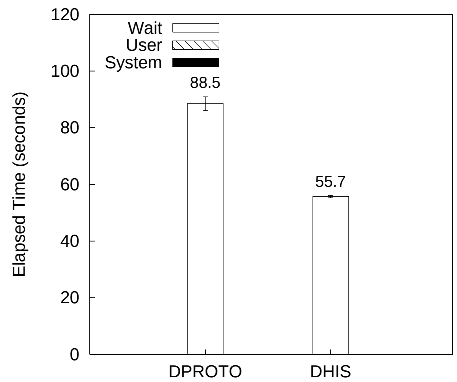


(d) Random Write Benchmark

Figure 6.5: *Microbenchmark results for DHIS. For each benchmark we show the time taken for regular DPROTO directly over RAID1 and RAID5, and compared them with DHIS with access pattern attributes*

performance (as meta-data blocks will mostly be accessed at random). By placing meta-data blocks in RAID1, DHIS has better sequential write characteristics, as random meta-data updates (such as updating the inode) gets absorbed by RAID1 while writing to a large sequential file on RAID5.

Therefore, by setting appropriate attributes about access patterns, DHIS can be made to perform substantially better than traditional storage systems that place data without the knowledge of their access patterns.



r

Figure 6.6: *Postmark results for Ext2DHIS over DHIS with selective NVRAM caching enabled compared to Ext2DHIS over DPROTO*

6.5.5 NVRAM Caching

To evaluate the benefits of caching selected candidates in NVRAM, we compared DHIS with regular DPROTO with file systems mounted in synchronous mode. DHIS choose all meta-data blocks as candidates for NVRAM caching and hence for a synchronous workload most of the meta-data block writes will be absorbed by NVRAM. Figure 6.6 shows the benefits of selective NVRAM caching. For this run, we configured Postmark to create 1000 files with sizes ranging from 10KB to 20KB, and 2000 operations. We used this smaller configuration as we ran this workload with synchronous mount of the file system. As seen from the figure caching meta-data selectively in NVRAM can improve write performance significantly (37%)for random I/O-intensive workloads.

Chapter 7

Case Study: Secure Deletion

In this section we describe our next case study: a disk system that automatically performs secure deletion of blocks that are freed. We begin with a brief motivation and then move on to the design and implementation of our *Secure Deletion Type-Safe Disk* (SDTSD).

7.1 Motivation

Data security often includes the ability to delete data such that it cannot be recovered [9, 30, 61]. Several software-level mechanisms exist today that delete disk data securely [40, 58]. However, these mechanisms are fundamentally insecure compared to disk-level mechanisms [69], because the former do not have knowledge of disk internals and therefore cannot guarantee that deleted data is overwritten.

7.2 Design

Since a TSD automatically tracks blocks that are not used, obtaining liveness information about blocks is simple as described in Section 3.2. Whenever a block is garbage collected, an SDTSD just needs to securely delete the block by overwriting it one or more times. The SDTSD must also ensure that a garbage collected block that is not yet securely deleted is not re-allocated; an SDTSD achieves this by deferring the update of the ALLOC_BITMAP until a block is securely deleted.

To improve performance, an SDTSD overwrites blocks in batches. Blocks that are garbage collected are automatically added to a secure-deletion list. This list is periodically flushed and the blocks to be securely deleted are sorted for sequential access. Once a batch of blocks is overwritten multiple times, the ALLOC_BITMAP is updated to mark all those blocks as free.

7.3 Prototype implementation

We extended our prototype TSD framework described in Section 3.5 to implement secure-deletion functionality. Whenever a block is garbage collected, we add the block number to a list. An asynchronous kernel thread wakes up every second to flush the list into a buffer, sort it, and perform overwrites. The number of overwrites per block is configurable. We added 403 lines of kernel code to our existing TSD prototype.

7.4 Evaluation

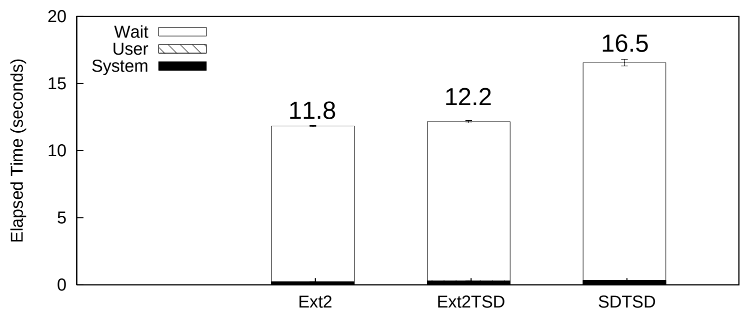


Figure 7.1: *Unlink micro-benchmark results for SDTSD*

To evaluate the performance of SDTSD, we ran an unlink micro-benchmark. Figure 7.1 shows the results of this benchmark. The I/O overhead of SDTSD over Ext2TSD was 40% compared to regular Ext2, mainly because of the additional I/O caused by overwrites for secure deletion.

Chapter 8

Context-Aware I/O Infrastructure

In this chapter, we present the concept of *Context-Aware I/O* (CAIO), a simple and generic way for applications to convey arbitrary information about their I/O behavior and relationships, without worrying about how the information will be used by the storage stack. In CAIO, an application-level *context* is propagated along with an I/O operation across the entire storage stack, in an end-to-end fashion. An application-level context is represented by one or more *context identifiers*. For example, a database application can have a unique identifier that it can propagate along with every I/O it generates, such that any storage layer can easily group all I/O generated by the database application. This also enables the lower layers of the storage stack to associate the data corresponding to the I/O with higher-level contexts and easily track the application's working-set.

In addition to working-set identification, application contexts also enable a new class of functionality that uses application-I/O relationships, such as easy and flexible performance isolation in large-scale distributed storage, and access-pattern aware caching and prefetching within the storage hardware.

To make CAIO a generic framework, we decouple the *generation* of application-level information from how the information is *used* within the storage stack. Most hint-based proposals to address the problem of information-gap in the past have tied these together. For example, in hint-based prefetching systems, the application provides hints of its future access, but the hints are specifically designed with prefetching in mind. The problem with such function-specific hints is that they require coordination and agreement between the layers involved. In a multi-vendor setup, such coordination translates into industry-wide consensus on the interface, a standardization process that takes years. In addition, such an approach cannot scale in an end-to-end manner to the multi-layered storage stacks that we have today.

Decoupling the generator and consumer of the context information leads to an interesting challenge: when the application could conceivably use more than one possible granularity of grouping I/O, how can it decide which one to use while being oblivious to how the grouping is interpreted by the lower level? For example, a database application can group the I/O requests it generates based on the database user, session, transaction, or query on behalf of which the I/O is issued; but the lower layers are oblivious to the granularity of the context. To solve this issue, contexts in CAIO are *hierarchical*. With hierarchical contexts, higher layers can encode multiple granularities of grouping, and the lower layers can decide which granularity is the best for the particular functionality that they provide.

Even in a hierarchical context, individual levels in the hierarchy remain completely opaque to the storage stack. For implementing functionality that needs more information about what these levels in the context mean, contexts can be annotated *offline* at any specific layer. In such cases, CAIO contexts will be used only as naming-identifiers to associate higher-level semantics.

We illustrate the generality and power of the context abstraction by prototyping and evaluating two case studies. Our first case study is an automatic working set identifier, *WorkSIDE*, which operates at the block-based storage hardware layer. WorkSIDE automatically tracks the data working set required for an application context to run to completion. WorkSIDE correlates contexts with the I/O and the corresponding data they access, thus obtaining a complete view of the entire set of data items that the particular application context requires. This working set can then be preloaded as appropriate in order to improve performance and availability, or to enable power optimizations. The second case study is a context-aware cache-placement algorithm within the disk that automatically tracks which application-level contexts exhibit sequential streaming access pattern and avoids caching requests with that context. We demonstrate the usefulness of both of our case-studies using prototype implementations we built for the Linux kernel, and evaluate various workloads.

The rest of this chapter is organized as follows. In Section 8.1 we discuss the utility of CAIO by presenting a few potential applications. In Section 8.2 we present a taxonomy of the various kinds of contexts in storage. We detail how we generalize the CAIO interface in Section 8.3. In Section 8.4, we describe CAIO design and application support.

8.1 The Utility of Context-Aware I/O

In this section we describe several usage scenarios that motivate tracking context information in the different layers of the storage stack. Many of these utilities cannot be implemented effectively without explicitly propagating application-level contexts. In Sections 9 and 10, we demonstrate our implementation of the first two usage scenarios described below.

Working-set Aware Features Identifying working sets of data for individual applications at the lower layers of the storage stack, enables interesting functionality such as application-aware prefetching [60], power-savings [87, 90], selective recovery of failed hardware [50], and improved data availability [70]. We describe our implementation of a disk-level working-set identifier and its usefulness in detail under Section 9.

Adaptive Caching and Prefetching The efficacy of caching and prefetching depends on the ability to identify access patterns. Context can enable caching and prefetching mechanisms to adapt their policies based on access patterns. Section 10 describes our implementation of a context-aware disk-level caching mechanism.

Application-Aware Performance Isolation Scheduling algorithms at different levels of the storage stack can leverage application-level contexts in scheduling decisions. For example, fair share disk schedulers can enforce fairness based on higher level logical tasks as against OS processes. Application-based resource isolation has been previously explored in the context of a single OS in Resource Containers [22]. Contexts can enable flexible resource isolation in an end-to-end fashion even in distributed storage.

Optimized Data Layout File systems can use higher level contexts as hints for optimal data placement on disk. Co-locating files and directories created in the same context could be beneficial under certain scenarios to achieve better spatial locality during reads.

Improved Accounting Context information associated with I/O operations can greatly help in I/O trace analysis. Trace analysis for resource consumption can be more accurate when it makes use of logical contexts pertaining to precise higher-level tasks. Contexts can also provide valuable hints about the dependencies of I/O operations and the causal relationships between them, for trace-based intrusion detection systems [43].

8.2 Context Types

Context in storage is quite useful as seen from the kind of functionality it enables (described in Section 8.1). We now define *context* as follows: *A context in storage is a reference or identification used to group, on some basis, several I/O operations or data.*

We now describe the types of contexts that are relevant to storage.

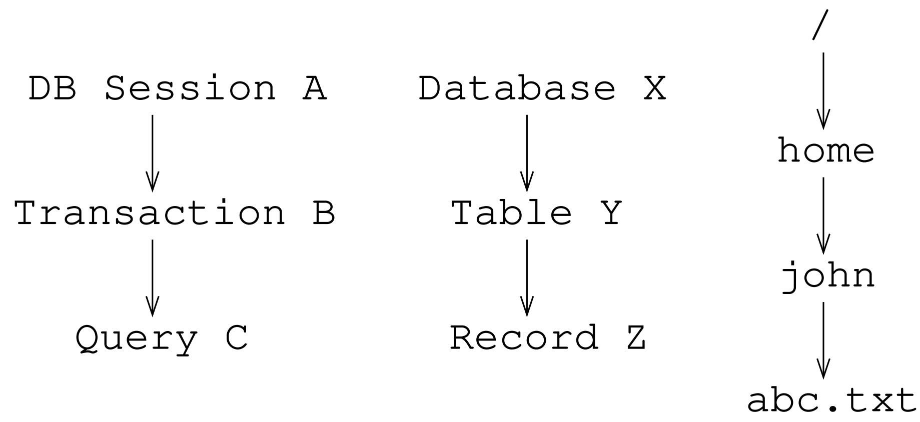
8.2.1 Data-bound vs. Access-bound

The two primary entities in storage are (a) data, and (b) I/O operations on data. Context in storage is mainly used for grouping several such data items or I/O operations. Therefore we classify context in storage broadly into two types: data-bound and access bound.

A context is said to be *data-bound* if it can be used to group several data items stored on disk, based on some metric. This grouping is independent of the way the data is accessed. For example, a data-bound context can group all blocks belonging to the same database table or file. Data-bound contexts can group data based on arbitrary criteria such as logical abstractions (files, directories, database tables, etc.), owning application or user, security domains, and so on. Data-bound contexts can be used to communicate higher-level data-structures to the disk, and enable functionality such as fault-isolated placement in RAID [70].

Note that the notion of data-bound contexts is similar in concept with other abstractions such as type-aware storage (Section 3) or object-based interface [53]. These other abstractions can be used as an alternative to data-bound contexts.

Access-bound contexts relate operations rather than the data pertaining to them. For example, an access-bound context can group all block write operations resulting from a single database query. Access-bound contexts enable new functionality that solely depend



(a) (b) (c)

Figure 8.1: *Examples of how hierarchical contexts can be constructed. (a) shows an access-bound context hierarchy. (b) and (c) show data-bound context hierarchies.*

on the characteristics of individual I/O requests. The caching and prefetching functionality described in Section 8.1 requires access-bound contexts.

Figure 8.1 shows a few examples of context hierarchies. Figure 8.1(a) shows a possible access-bound hierarchy for a database application. Figures 8.1(b) and 8.1(c) show data-bound context hierarchies that communicate data abstractions.

8.2.2 Repeatable vs. Non-Repeatable

The lifetime of a context identifier is defined by the application that generates it. When a single context identifier is used every time to refer to a particular logical context, we call it a *repeatable* context. For example, when a context is used to group files within an access-control domain, the same identifier has to be reused every time when operations are performed on that domain. Applications have to generate such contexts using a deterministic method and may maintain persistent states to track contexts.

Non-repeatable contexts have transient identifiers. For example, if a `pid` is used as a context identifier to group I/O operations generated by a particular program, every time the program runs, the identifier becomes different, although the logical context remains the same. Non-repeatable contexts do not require any state to be maintained at the application-level.

8.3 Generalizing the Interface

In this section, we describe how we can cope with arbitrary context generation process at the application-level, and achieve independence between the generation and usage of application-contexts. We also describe how lower layers of the storage stack can extend contexts or correlate across different context types.

Hierarchical Contexts To achieve generality in the CAIO interface, the context generation process at the application-level must not make any assumptions about how the lower layers use the context. However, at the application-level, there may be several different ways to generate a context, each useful for different kinds of functionality at the lower layers. A single application-wide context identifier an be used to easily group all data required by the application, whereas more fine-grained context identifiers within an application help communicate different streams of I/O requests generated by sub-components of within same application. For example, a single DBMS-wide context can be used to group all I/O and data that the DBMS manages. This enables functionality such as working-set identification for the entire DBMS. On the other hand, a per database session-level context can be used for easy performance isolation between database user sessions. We use the term *context granularity* to refer to the different possible ways to generate contexts within an application.

Therefore, for generalizing the interface without hampering the kind of functionality it enables, we evolve a context scheme where the application can encode all possible granularities as a single context, passing down *context hierarchies* (for access-bound and data-bound) rather than a single identifier. For example, a DBMS can generate access-bound contexts in granularities such as sessions, transactions, and individual queries, and data-bound contexts in granularities such as databases, tables, and records.

Lower layers of the storage stack can use hierarchical contexts without making assumptions about what each of the levels in the hierarchy mean. For example, a caching layer that wants to classify some context to exclude caching (e.g., sequential contexts) can track the statistics on sequentiality at each level of the context hierarchy, and then choose the highest level that exhibits homogeneity in the access pattern. Depending on the specific behavior the layer is looking at (e.g., sequentiality, correlated access of the same pieces of data), the definition of homogeneity changes. Hierarchical contexts enable decoupling the application from worrying about which behavioral properties the lower layers are interested in; instead the application just conveys its state, and the lower layers make their independent decisions on the notion of homogeneity they care about, based on the layers' own per-context statistics.

Note that for a context hierarchy chain in CAIO to be meaningful, every context in the chain should qualify a logical subset of the access or data domain qualified by its parent context. For example, a per-query context identifier can be a child of the transaction identifier in which the query is a part. However, a context identifier that qualifies the class of *all* `select` queries in a DBMS cannot be a child of any particular transaction identifier, as `select` queries can be part of any transaction.

Annotating Contexts with Semantics Certain functionality may require more information about what each level in the hierarchy means, at some specific layer in the storage stack. For example, a context-based proportional-share disk scheduler needs share proportions to be associated with levels in the context hierarchy. For this purpose *offline* mechanisms can be used to annotate context identifiers with functionality specific information. For example, applications can co-ordinate with a specific file system through offline *ioctl*s to associate locality hints with stored context identifiers. Note that these annotations are not part of the CAIO infrastructure, but can be done separately between any two layers that needs to coordinate to implement a specific functionality. In the example of a proportional-share disk scheduler, the application and the disk scheduler need to co-ordinate offline to annotate context levels with share proportions.

Context Transformation With hierarchical contexts, any layer in the storage stack can add new levels to the context chain, as long as the subset invariant is preserved. For access-bound contexts, the subset relationship is maintained as an operation propagates from top to bottom. For example, a *select* query generated from a database gets transformed into one or more file read operations at the file system, and then further transformed into several block read operations at the device driver or the disk level. Therefore, any layer in the stack can add new levels to communicate grouping of sub-operations at their level.

However, for data-bound contexts, subset relationship is harder to ensure across layers. This is because the data abstractions used by higher layers in many cases are not supersets of the lower level abstractions. For example, an application can store several B-trees within a single file, and hence there is no subset relationship between the abstractions used by this application and that of the file system. Therefore generic transformation of data-bound contexts across layers is harder to achieve; but lower layers can associate new data-bound context hierarchies with I/O, if the application does not pass a data-bound context. We impose a constraint that intermediate layers should not add new levels to data-bound contexts, unless the higher-level layer did not specify a context of its own.

Correlating Across Context Types Data-bound and access-bound contexts passed by the application can be completely independent of each other and need not necessarily indicate association between the operation and the data it operates. This makes generation of contexts at the application-level much less complicated. However, lower layers that use these contexts can maintain their own history information of contexts, and correlating data-bound and access-bound contexts. Correlating context types enables useful functionality. For example, identifying the working set of data accessed by an access-bound context can be useful for implementing interesting optimizations as described in our first case-study detailed under Section 9.

8.4 CAIO Design

End-to-end association of context with I/O requires passing application-generated context with every I/O operation throughout its lifetime. We evolve a framework through which context can be passed from an application all the way down to the storage hardware (e.g., a disk). In this section, we describe the changes required to the storage stack and user applications, to support contexts.

We propagate context in the storage stack by means of *context objects*. A context object contains upto two context chains, one each for data-bound and access-bound types. These context types are based on the discussion under Section 8.2. Context objects also carry information about the repeatability of the context chains. Repeatability is at the granularity of an entire chain and not the individual context identifiers within a chain. The structure of a context object is shown in Listing 8.1.

```
struct caio_context {
    int data_bound[MAX_DATA_LEVELS];
    int access_bound[MAX_ACCESS_LEVELS];
    short data_levels;
    short access_levels;
    int flags;
};
```

Listing 8.1: Structure of a context object. The fields data_levels and access_levels indicate the number of levels in the data and access-bound context chains. Flags contain information about repeatability and inheritance properties (Section 8.4.1) for the context.

8.4.1 Associating Contexts With I/O

The CAIO framework contains a user library that exports routines to construct context objects and add new levels of hierarchy to existing context objects. User applications can generate context objects through these routines and associate them with I/O operations. Our framework provides three different ways for user applications to associate contexts with I/O operations. They are, (a) an extended system call interface (b) group contexts and (c) context inheritance. We detail each of these mechanisms below.

An Extended System Call Interface We have an extended system call interface that passes context objects along with storage primitives such as `open`, `read`, `write`, `unlink`, etc. Each of these I/O system calls include an additional argument for the context object. The framework also includes a wrapper library for user applications to call these new system calls. Listing 8.2 shows an usage scenario for the extended system call interface. Note that when there is a single context object that needs to be passed for several system calls, *group contexts* can be used for better performance, as described below.

Group Contexts For applications that need to perform a several I/O operations with a single context object, we provide a new system call for setting and unsetting contexts into the kernel. The scope of this association is just the specific thread of execution. Therefore applications can first set a context and then issue any number of regular I/O system calls (such as `open` or `read`), and the corresponding context object will be associated with every operation.

Context Inheritance To support easy usage of contexts in cases where the smallest granularity is a process, our framework includes a context inheritance mechanism using which any process can set an *inheritable context* into the kernel. All child processes and threads of such a process will then inherit the same context hierarchy. We developed this feature so that there would be no modifications required to applications whose lowest context granularity is a process. For example, if a project compilation task requires several applications such as gcc, ld, binutils etc., the entire compilation task can be run through a shell that has an inheritable context set, instead of modifying every application to pass contexts.

```
int fd; char buf[128];
struct caio_context *context;

/* Allocates and sets top-level databound
 * and accessbound identifers as 1 */
context = caio_create_context(1, 1);

/* Adds a new level to the access/data
 * hierarchy with identifier 2 */
caio_add_level(context, 2, 2);

/* CAIO system call interface */
fd = caio_open("/home/joe/abc.txt",
          O_RDONLY, &context);
err = caio_read(fd, buf, 128, &context);
caio_close(fd, &context);
```

Listing 8.2: Passing contexts from the user-level using the CAIO extended system call interface. Note that in this case group context (described in Section 8.4.1) can be used as well, because a single context object is used for all calls.

8.4.2 Context Propagation

In CAIO, each layer receives contexts from the layer above and passes it to the layer below after using them if applicable. Note that a single operation at a particular layer could translate into multiple operations in the layers below. For example, a file create operation at the file system level could result in multiple block write requests to the device driver. Therefore it is each layer's responsibility to propagate context objects appropriately to the layer below. In cases where there are more virtualization layers such as software RAID or logical volume managers (LVMs), such layers should be aware of contexts and propagate them below. Any layer can choose to store contexts in its own structures for its needs, before passing them down.

Hardware Interface Extensions To propagate contexts end-to-end, we extend storage hardware interfaces to pass generic context objects along with every I/O request. For example, the SCSI/IDE read and write primitives take context objects. There are a number of proposals in the past that suggest interface extensions to disk systems for communicating higher-level semantic information [17, 49, 53, 68]. We believe that the generality of the CAIO interface would make it easier for disk vendors to adopt.

Dealing with Operation coalescence Multiple logically independent I/O operations may be coalesced into one at any layer in the stack. For example, multiple file write operations to the contents of the same file block could result in a single block I/O at the disk level due to write buffering. To handle such cases, we support multiple context objects to be associated with a single lower level I/O. Layers that receive these contexts must process them one by one as if they were from different I/O operations.

Storing Contexts Repeatable contexts may need to be stored by layers to implement optimizations that involve tracking context history, or correlating different context types. We developed a `context-store` in-memory data-structure as part of our framework to enable easy storage of context hierarchies at any layer of the storage stack. A context store manages context hierarchy in a tree structure in which each node represents a context identifier of a specific level in the hierarchy identified by its depth in the tree. Each tree node also includes a *private data* field where information about that specific chain can be stored. The context-store structure provides primitives for common operations such as adding new chains and updating private data.

8.4.3 Linux Implementation

We implemented our CAIO framework in the Linux kernel 2.6.15. We added new system calls for context-aware file I/O operations and implemented a user-level library for applications to easily use the new system call interface. The new system calls allowed context objects to be passed with `open`, `read`, `write`, `pread`, `pwrite`, `close`, `mkdir`, `unlink`, `rmdir` and `readdir` operations. We modified the following objects to add a new field to store contexts. (a) `task_struct` which represents a running process or thread. (b) `buffer_head` which represents a block buffer in memory. (c) `bio` which represents an I/O to a block device. The `buffer_head` and `bio` objects can optionally contain a list of contexts during operation coalescence.

We implemented the new system calls as wrappers to the unmodified system call handlers for the operations. The wrapper system calls set the context object in the `current` task object before calling the unmodified handlers. Note that the wrapper calls unset the context upon completion of a system call, so that the scope of a passed context would be just that system call. The different layers in the OS that service the I/O operation use the context object from the `current` task object and propagate it to the corresponding `buffer_head` and `bio` objects appropriately. As the `task_struct` object is unique to a particular process or thread, this method works for multi-process workloads as well.

For group contexts, we added a new system call which assigns or removes the corresponding context in the current `task_struct` object. For inheritable contexts, we modified the `fork` system call to copy the context object of the parent, to the forked process. We also implemented the context-store data-structure as part of the kernel so that any layer such as the file system or device driver can maintain its own store.

Overall, the modifications required to implement the CAIO framework were small. We added only 350 lines of new kernel code and 150 lines of user-level code.

8.4.4 Application Support

The method of generating contexts at the application-level depends on specific application architectures. In general, if an application can classify its activities into distinct logical tasks, and (or) if it can group data it uses based on some criteria, it can generate contexts in a meaningful manner. Based on the kind of application, the granularity and type of contexts it can generate can vary. Some low level applications such as Unix utilities (e.g., `ls`, `cat`, etc.) can just provide an interface to the caller to pass contexts (e,g., command line arguments). We have modified some basic utility programs such as `cp`, `cat`, and `ls` to accept contexts as command line arguments. This enables a higher level caller application (e.g., a shell script) to group all its operations under the same context.

Context-Aware MySql We have modified the MySql DBMS [57] with InnoDB [38] as the storage engine, to generate and propagate contexts at various granularities. MySql has the notion of database client connections which can obtain service from the DBMS. Each client connection gets serviced by a separate MySql thread, and can run several transactions and queries. We modified MySql to pass contexts at three granularities in the form of a hierarchy: connection-level, transaction-level, and a single query-level. Overall the modifications required to propagate contexts across the various layers of MySql and InnoDB were simple. We added only 30 lines of new code and modified 345 lines of existing code, mostly for passing an additional argument for a number of functions. We use our Context-Aware MySql as an application to evaluate our framework and some of the case-studies described in Sections 9 and 10.

8.4.5 Evaluation

We evaluated the overheads associated with passing context objects across the storage stack for all file system operations. In this section we first describe our test setup and the details of the experiments we ran. Note that the setup described in this section applies to all our benchmarks presented under Sections 9 and 10 as well.

We conducted all tests on a 2.8GHz Xeon with 1GB RAM, and a 74GB, 10Krpm, Ultra-320 SCSI disk. We used Fedora Core 6, running a Linux 2.6.15 kernel. To ensure a cold cache, we unmounted all involved file systems between each test. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student-t distribution. In each case, the half-widths of the intervals were less than 5% of the mean.

Experiments

In this section we describe the set of experiments and their configurations that we used for evaluating the CAIO and the case-studies.

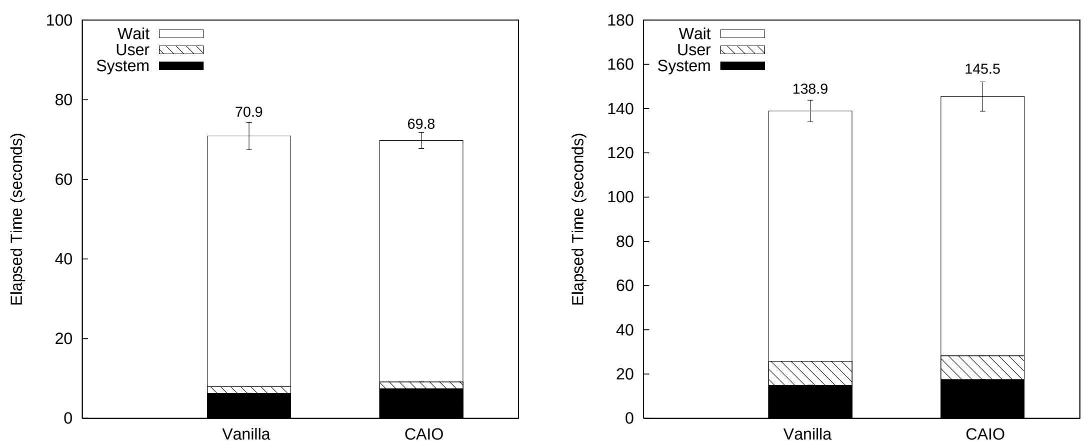
Postmark For an I/O-intensive workload, we used Postmark [82], a popular file system benchmarking tool. Postmark stresses the file system by performing a series of file system operations such as directory lookups, creations, and deletions on small files.

	Regular Response Time (s)	CAIO Response Time (s)
Delivery	0.096	0.109
New Order	0.039	0.064
Order Status	0.033	0.29
Payment	0.000	0.000
Stock Level	0.169	0.524
Throughput (tpmC)	67.13	64.35

Table 8.1: *TPC-C Benchmark results for the CAIO framework*

TPC-C TPC-C [79] is an On-Line Transaction Processing (OLTP) benchmark that performs small 4 KB random reads and writes. Two-thirds of the I/Os are reads. We set up TPC-C with 50 warehouses and 20 clients. We compare our context-aware MySql running on our CAIO framework with regular MySql running on a vanilla kernel. The metric for evaluating TPC-C performance is the number of transactions completed per minute (tpmC). We report tpmC numbers for each benchmark.

Results



(a) 50,000 Operations (b) 100,000 Operations

Figure 8.2: *Postmark Results for CAIO Framework*

Figure 8.2 shows the overheads of our CAIO framework for Postmark for two different number of operations. As seen from the figure the overall elapsed time overheads were small (2% to 4%) compared to regular I/O. This overhead is mainly because of the additional user-to-kernel copies for communicating context objects from applications.

TPC-C Results The TPC benchmark results for regular MySQL and our modified context-aware MySQL ran over the CAIO kernel is shown in Table 8.1. The workload loads tables into a Mysql server at start-up and runs a mix of queries on these tables for a user defined time. We configured the benchmark to run with five warehouses and created two client connection which ran queries on all five warehouses for ten minutes. As seen from throughput and response time numbers, overheads of the CAIO framework is quite small.

Chapter 9

Case Study: Working Set Identifier

Our first case study is the automatic **Work**ing **S**et **IDE**ntifier (*WorkSIDE*). WorkSIDE that uses both access-bound and data-bound contexts to automatically infer the minimum set of data items required to be available in order for an application (or a specific instance of an application) to run to completion.

9.1 Motivation

This ability to accurately identify working sets of application contexts at a fine grained level has various kinds of applications.

Performance The working set of the application can be preloaded into a much faster but smaller memory hierarchy (e.g., a flash storage layer that provides about 100x better random access read performance), thus essentially shielding the application from performance variability due to disk access.

Availability WorkSIDE enables fault-isolated placement of application working sets enabling truly graceful degradation during multiple disk failures similar to D-GRAID [70]. While D-GRAID could just co-locate files or directories, WorkSIDE can co-locate higher-level application working-sets within failure domains.

Power Savings Many recent systems have looked at saving power by switching off a subset of disks in a large RAID array in such a way that applications can still function properly without the switched-off disks [87, 90]. These systems go to great complexity to identify the subset of data that is currently under use, yet these techniques are most often approximate and too coarse-grained. Being more informed about the application's access patterns and data abstractions, WorkSIDE can do a better job at such power optimizations by being more aggressive and more accurate.

Disconnected operation Another usage scenario for WorkSIDE is when the user wants to preload the working set for a specific application context in local storage for disconnected operation, say, in a mobile environment. This enables Coda-like hoarding [44], but can be much more accurate, fine-grained and automated. For example, if the user works only on a specific build target in a large body of source code, just the subset of source files (and the metadata) needed for the target can be automatically preloaded to local storage.

The key to WorkSIDE is its ability to correlate a repeatable access context with the data context it accesses. WorkSIDE achieves this by associating with each node of the access context hierarchy, the aggregated set of data items that are accessed by that context. Semantic aggregation of such data is possible because data-bound contexts are hierarchical in nature conveying data abstractions in several granularities (such as files or directories). Tracking working set at an aggregated level enables much simpler and reliable tracking of repeatability. For instance, if an application touches different parts of a file in its different runs, block-level tracking may not find much of a repeatability, whereas tracking at the file-level would indicate the pattern. Since the data context hierarchy essentially contains information of the entire data abstraction tree, it can track this information at various granularities, and decide on which granularity provides the best trade-off between the amount of data to be preloaded and ensuring completeness for the application.

9.2 Design

To determine the working set of a higher level logical task, WorkSIDE has to track history of both data-bound and access-bound contexts for every task. We designed WorkSIDE as an on-disk mechanism to demonstrate its working as part of the firmware of a high-end block-based RAID storage system. WorkSIDE can potentially exist at any layer of the storage stack such as the file system or the device driver. Through our design, we show that even in the lowest layer of the storage stack (the storage hardware), working set identification can be done to an acceptable level of accuracy, through context-aware I/O.

For WorkSIDE to correctly determine the working set of data for a given access-bound context, the higher application has to pass data contexts to communicate the semantic organization of data. This can relate to on-disk structures such as B-trees, database tables, files, and directories. In this section, we first detail how access-bound contexts can be associated with corresponding data-bound contexts. We then discuss a few policies that can be adopted to determine the granularity of the working set of a given context. Lastly, we present our prototype implementation of WorkSIDE.

9.2.1 Associating Access with Data

WorkSIDE maintains two context stores (described in Section 8.4) to track access-bound and data-bound contexts respectively. Each store has context trees to represent the hierarchy. We call tree nodes in the access and data stores as *Access-Context Nodes* (ACNs) and *Data Context Nodes* (DCNs) respectively. Note that, as data-bound context is mainly used to communicate the semantic structure of data, it need not necessarily be passed by the higher-level application for every I/O request. For example, if a DBMS uses the `table` and `record` abstractions as data-bound contexts, it may pass the context hierarchy only when such abstractions are created (e.g., a table creation) or updated (e.g., a new record insertion). For example, the DBMS need not pass data-bound contexts for every `select` query. To handle this condition, WorkSIDE may have to map access-bound contexts accompanying a block I/O request with a pre-existing data-bound context hierarchy.

The following are the contents of a DCN: (a) A context identifier. (b) The number of blocks in the entire sub-tree with the node as root. (c) A list of block numbers associated with the context (if it is a leaf node). Every time a block I/O has an accompanying data-bound context chain, the corresponding block number is added to the leaf DCN of the chain. (d) A list of pointers to its child nodes. (e) A back-pointer to its parent node. This is used to increment the number of blocks in every parent along the chain when there is a new addition to a leaf node.

While adding a node to the tree, we enforce the *single parent* constraint, where every node must have at most one parent. When there is a context chain passed, that violates this condition, we truncate the chain after the spurious node while adding it to the tree. In almost all common cases, this would not affect the accuracy of the data-bound context tree, as most data-abstractions already follow this rule. For example, a single block cannot belong to more than one file (except in rare cases such as hardlinks in Ext2).

WorkSIDE also maintains a hash table, BDTABLE, to map block numbers to the corresponding leaf nodes in the data context tree. The BDTABLE is used to lookup the data context for any block when an I/O request to it does not have an associated data-bound context. Upon receiving a block I/O request with a access-bound context, WorkSIDE can map the corresponding block number to any level of abstraction in the data-bound hierarchy by just traversing through the parent back-pointers in each node in the data context tree.

In the next section, we describe how this infrastructure is augmented with association policies to determine the optimal granularity of associating a data-bound working set for a given access-bound context.

9.2.2 Working Set Identification

Identifying the working set for a given node in the access-bound context tree involves associating that ACN with one or more DCNs. Therefore every ACN in the access store contains pointers to one or more DCNs.

Greatest-Common-Prefix Mode We designed WorkSIDE to operate under two different modes for choosing the appropriate DCN for a given ACN. In the first (and simple) mode, which we call the *Greatest Common Prefix* (GCP) mode, WorkSIDE maintains utmost one DCN per ACN. Whenever there is an I/O in the context of an ACN, the request block number is looked up in the BDTABLE to find the leaf DCN to which the block number is associated. The leaf DCN is associated with the ACN if the ACN did not previously have a DCN associated. If not, the greatest common prefix node in the tree (starting from the root) for the new leaf DCN and the previously associated DCN is computed (using the parent back-pointers) and associated with the ACN. The working-set is enumerated by just traversing the sub-tree starting from the associated DCN. This method of enumerating the working set for an ACN ensures completeness, but under some scenarios there could be a significant number of falsely associated blocks. For example, if an access context A

reads files /home/john/plan.txt and /home/john/private/list.txt, the GCP method of association would include the entire contents of /home/john/ in the working set of A. A variant of the GCP mode mitigates this problem under some scenarios by tracking the longest depth to traverse while enumerating blocks, along with the ACN. With this, the working set of A would just include files up to depth level 3 (/home/john/private).

Multi-DCN Mode In the second mode, which we call the *Multi-DCN mode*, WorkSIDE tracks a list of DCNs per ACN. Every ACN has a list of duplicate eliminated pointers to parent DCNs. To enumerate the working set for a given ACN, the following procedure is used: for each DCN associated, all blocks belonging to their immediate children are included. For example, if an ACN B reads files /home/john/plan.txt and /home/john/private/list.txt, DCNs for /home/john and /home/john/private will be associated with B. While enumerating the working set of B, all files (not sub-directories) under /home/john and /home/john/private will be included. Therefore, the multi-DCN mode of association provides more accurate identification of working sets. However, this method needs to track more information per ACN. In the procedure described above, we choose the hierarchy one level above the leaf DCN for every block access. However, the number of such levels can be configurable based on specific system and workload requirements.

WorkSIDE can also track information required for both GCP and multi-DCN modes simultaneously (every ACN can have both the list of parent DCNs and a single GCP node). Based on the kind of usage scenario for the working set, enumeration process can be decided dynamically to choose the optimal granularity.

9.2.3 Prefetcher

We developed an on-disk prefetching tool that uses WorkSIDE to enumerate the working set of access-bound contexts and prefetch them into a faster storage. For prefetching, we tracked the repeatability of the working set of each ACN, and for repeatable ACNs, we prefetch and serve the entire working set from the faster storage medium. Currently we use RAM to cache prefetched working sets, but this could even be a fast secondary storage device such as flash. While deciding whether to prefetch a working set, we take into consideration the size of the working set and the available space in the prefetch cache. In our design, we use a simple scheme where we prefetch working sets less than half the size of the prefetch cache subject to remaining space availability in the cache. More advanced algorithms such as best-fit and worst-fit can also be implemented to decide the appropriate working sets to prefetch.

To evaluate our working-set aware prefetcher, we compiled several modules in the Linux kernel source, and e2fsprogs package [80], with inheritable contexts. We found that once working-sets were identified by WorkSIDE and prefetched into RAM by our prefetcher, there were no requests sent to the disk during the compile workload. Therefore, working-set aware prefetching of data enables turning off disk drives (and hence save power) in the case of repeatable workloads.

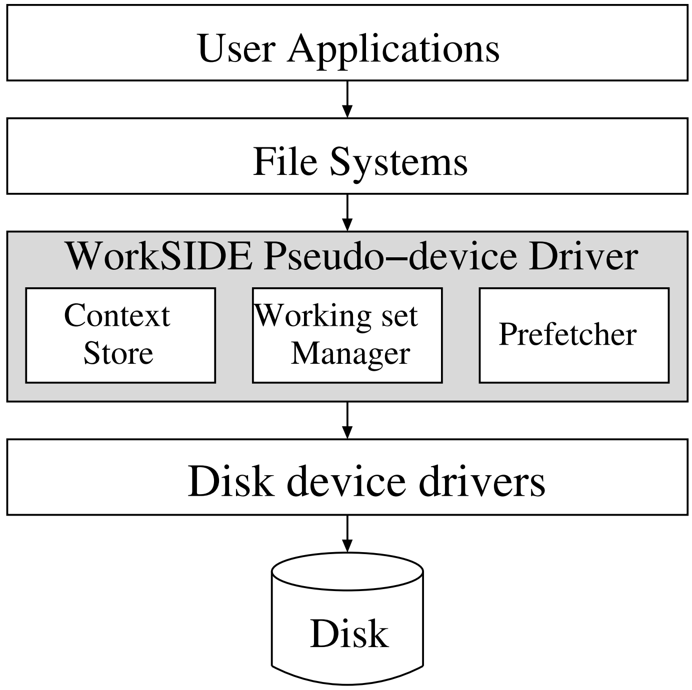


Figure 9.1: *Prototype implementation of WorkSIDE in the Linux kernel. The prefetcher component in WorkSIDE prefetches common working-sets into memory to save power.*

9.3 Implementation

We implemented a prototype of WorkSIDE and our prefetching tool as a pseudo-device driver in Linux kernel 2.6.15 that stacks on top of an existing disk block driver. The pseudo-device driver receives all block requests, and redirects the common read and write requests to the lower level device driver, after storing context information that needs to be tracked. Our prototype of WorkSIDE included both the GCP and multi-DCN modes of associating data-bound contexts. It contains 3020 lines of new kernel code. Figure 9.1 shows the architecture of our prototype.

For testing WorkSIDE, we also modified the VFS layer of the Linux kernel to encode the pathname of the entity being operated (file or directory) along with every lower level I/O request. File system meta-data blocks such as super blocks, bitmaps and directory blocks have to be dealt with separately, as they may not particularly belong to a specific application. To handle such blocks, we modified the Linux Ext2 file system to associate a generic "common" context which can be interpreted by any layer as one that is not associated with any particular access-bound context. We call our modified Ext2 file system, Ext2C.

9.4 Evaluation

We evaluated the correctness and performance of our prototype implementation of WorkSIDE. For correctness we used a Linux kernel module build process, and for performance,

Module	# Directories	# Files	# Blocks (4k)
Ext2	14	315	1149
Ext3	14	328	1452
ReiserFS	14	328	1432
NTFS	14	320	1769

Table 9.1: Compilation Working Set Statistics

we used the Postmark benchmark described under Section 8.4.

9.4.1 Completeness of the working-set

To verify the completeness of the working-sets identified by WorkSIDE, we implemented a prefetch cache layer beneath the file system that prefetches the working-set for selected access-bound contexts. We then simulated a disk crash by disallowing disk I/O from our pseudo-device driver, and repeated the workloads for the corresponding contexts. We performed this for kernel modules compiles and several micro-benchmarks, and in all cases the prefetch cache layer serviced all I/O requests. This shows that working-sets identified by WorkSIDE are complete.

9.4.2 Kernel Modules Build

Our goal during this test was to evaluate the correctness of the working set identification mechanism of WorkSIDE. We untarred a vanilla Linux 2.6.15 kernel on our Ext2C file system mounted over our WorkSIDE pseudo-device driver. We did this through a shell that has an inheritable access-bound context set (described under Section 8.4.1), with depth one. We then remounted the file system to eliminate cache effects and compiled the source-code of a few file systems (Ext2, Ext3, Reiserfs, and Ntfs) under the `fs/` sub-directory of the kernel source. While compiling each file system, we used different shells with different second-level inheritable contexts set. All compilations were done with the same top-level hierarchy of context, but for each compilation, the second-level was different. Therefore, we were able to track the working-set of each of the individual compilations. Note that we initialized the build process through "`make install`" separately at the beginning, and remounted the file system after each compilation. We ran this test over WorkSIDE for both GCP and multi-DCN modes of operation.

Under the GCP mode, we noticed that the working sets of every single file system compilation was identified as a the root of the kernel source tree. This is because, a file system module compilation would refer to files under `include/` and `fs/` and hence the greatest common prefix node becomes the root of the kernel source.

When we ran the test under the multi-DCN mode, we saw WorkSIDE identify separate working sets for each of the file system compilation contexts. Table 9.1 shows the total number of directories, files, and blocks associated with the working set of each compilation. We identified these by dumping the entire access-bound context tree of WorkSIDE and their associated DCNs. In each compilation context, the generated object files were

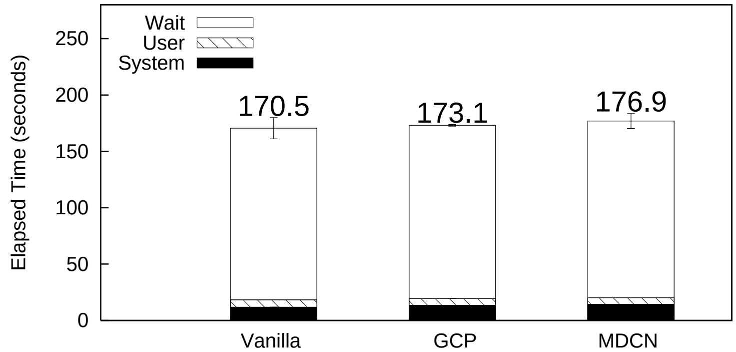


Figure 9.2: *Postmark Results for WorkSIDE (200 Sub-directories, 20,000 Files, and 200,000 Transactions.). This shows the overheads associated with the process of working-set identification at the disk-level.*

also included in the working set as the same inheritable context was passed for write operations as well.

We also used the Multi-DCN mode of WorkSIDE to calculate the working-sets for kernel compilation with `make allnoconfig` and `make allyesconfig`. For compilation using *make allnoconfig*, the size of the working-set came out to 32.6MB. For `make allyesconfig`, the working-set size was 3GB. As the object files during compilation are created from the same context, they were included in the working-set.

9.4.3 Postmark

To evaluate the performance overheads of WorkSIDE, we used an I/O-intensive benchmark, Postmark. We ran our modified Postmark that passes context objects with each I/O request, over WorkSIDE in its two modes, and compared it with regular Postmark running on top of a normal disk. For the regular Postmark we used unmodified Ext2 as the file system and for WorkSIDE evaluation, we used our modified Ext2C file system. Figure 9.2 shows the overheads of WorkSIDE compared to regular disks.

WorkSIDE under the GCP mode of operation had an elapsed time overhead of 1.5% compared to regular disk. The overhead mainly consists of system time (12%) caused because of updating context trees and tracking greatest common prefixes. Under the multi-DCN mode of operation the elapsed time overhead was 3.7% compared to a regular disk, caused by a 20% increase in system time. The increase in overheads compared to GCP mode is because under the multi-DCN mode, WorkSIDE has to track multiple data nodes per access-node. If WorkSIDE is implemented in a real disk, tracking context trees would be done by the disk firmware and hence would not incur the host CPU overheads.

Chapter 10

Case Study: Context-Aware Caching

Modern large-scale storage systems have hundreds of gigabytes of built-in main memory [16], primarily for caching purposes. However, today's storage systems cannot adapt their caching policies based on application-level workloads or data semantics, as they lack information about higher level semantics. This is caused by an excessively simple disk interface [18, 70]. Application-aware caching policies have been found to be quite useful in the context of OS level caches [14]. Yet today's disk systems cannot even separate independent I/O streams generated by two different applications, making it harder to implement application-aware caching policies.

In this section we design and evaluate *Context-Aware Cache* (CA-*cache*), an on-disk caching mechanism that differentiates independent I/O streams using logical contexts and tunes its caching policies based on individual access patterns.

10.1 Design

We designed CA-*cache* as an on-disk LRU write-through cache layer. The goal of CA-*cache* is to identify sequential streams of I/O and disable caching their data, as mostly sequential I/O streams do not benefit from read caching. As we are interested in the access-patterns to tune the caching policy, this application uses access-bound contexts.

Architecture CA-*cache* consists of a set of dynamically-built context trees and an LRU cache. Each tree represents a group of hierarchical contexts with the same root context. Each node represents the hierarchical context specified by the path from the root of the tree to that node. Context trees are created or updated on each read request that specifies an access-bound context.

Classification of Contexts Each node in the tree contains the following information about a particular context: (a) the inferred access-pattern for the particular context, (b) the block number for the last read I/O request required to track sequentiality, and (c) two counters that track the number of successive sequential and random read requests in the past. A context node is initialized as random-access upon creation. Based on the last read request and the current request, either the sequential or the random counter is incremented and the other is reset. When the values of the counters exceed a *threshold*, the node is classified as sequential or random as appropriate. Note that an already classified node could be re-classified when its access pattern changes. Upon receiving any read request, the counters in all nodes that are part of the current context are updated and the nodes are re-classified if needed. We call the number of sequential read request required for classifying a node as sequential, the *sequential threshold*. The sequential threshold is configurable, and can range somewhere between 10 and 100. A sequential-access node is re-classified as random upon a single out-of-order read.

Caching Methodology Our classification scheme allows for different hierarchy levels in the same context chain to be classified differently. For example, two sub-contexts that are part of the same parent may be doing sequential I/O in their own levels. However, since the I/O from the sub-contexts could be received interleaved, the parent would be classified as random. CA-*cache* does not require context identifiers to be repeatable. Therefore, it contains a mechanism to automatically forget contexts based on a timeout. We periodically purge context tree entries that represent inactive contexts (without any requests) beyond a time threshold.

10.2 Evaluation

We implemented a prototype of our on-disk caching mechanism as a pseudo-device driver in the Linux 2.6.15 kernel similar to WorkSIDE. We maintain the context trees in memory and an asynchronous kernel thread wakes up periodically to purge timed out context entries. If the block is present in the LRU cache, the pseudo-device driver services the request from the cache, thereby avoiding a request to the lower level. Otherwise, the request is directed to the lower level and the cache is updated on completion of the request, if the request belongs to a random-access context.

Read Micro-benchmark To evaluate CA-*cache*, we ran a micro-benchmark that generates synthetic random and sequential read workloads simultaneously and calculated the overall throughput of the random workload. We compared the throughput results of CA-*cache* with a vanilla LRU cache layer which treats all contexts equally. Both CA-*cache* and vanilla LRU cache used 4MB of cache (1,024 4KB pages) for this benchmark.

We ran a user program that generates workloads shown in Figure 10.1. The user program has four execution contexts (threads), A, B, C, and D which use their own files for I/O. Thread A reads a 4GB file sequentially with context {1-2-5} (see Figure 10.1). Thread B reads a 4GB file sequentially, but it uses contexts {1-3-7} and {1-3-8} for alternate reads. Thread C is identical to thread B, but it uses contexts {1-4-9} and {1-4-10}. Thread D reads random locations from a 4GB file using context {1-2-6}. For thread D, we use a random number generator that repeats itself every 1,024 reads. The threads run until any one of the sequential threads exits after reading 4GB of data. In our experiment, the throughput of the random workload when run under the vanilla LRU cache was 0.098 MB per second, whereas with CA-*cache*, the throughput was 7.71 MB/Sec.

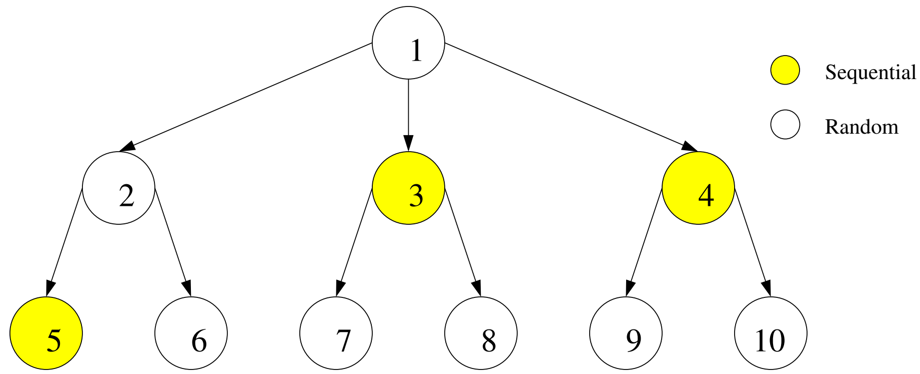


Figure 10.1: *Context tree used for CA-cache micro-benchmark. After our micro-benchmark, CA-cache classified the grayed nodes as sequential and the rest as random.*

MySQL Micro-benchmark For this benchmark, We created two identical tables SEQ and RAND in MySQL with 4,200,000 records each, and ran random and sequential query logs simultaneously. The tables were approximately 233MB in size. The sequential query log contained a `select *` query on the table. For a random workload, we selected a subset of the records at random and issued select queries based on their record IDs. To show the benefits of caching random streams alone, we repeated the random query log ten times. We also ran the sequential log in a loop till the random workload completed. We determined the throughput of the random workload (number of queries executed per second) while the sequential workload was running in parallel. It was 266.13 queries per second without selective caching, while it was 614.15 queries per second with selective caching.

Chapter 11

Case-study: Context-Based Disk Scheduler

Modern servers run hundreds or thousands of applications simultaneously. Inevitably some of these applications would have higher storage performance requirements compared to others. Critical applications using a shared storage system need to be insulated from the impact of transient workload surges caused by other applications. A well-known model to approach this problem is to allocate a proportional share of resources [72, 83, 84, 86]. Such an allocation of shares must be based on logical higher level tasks. However, today's storage stack cannot even distinguish between independent streams of I/O requests emanating from different logical tasks. Today, the file system can relate I/O operations with threads of execution (e.g., an OS process) while lower levels of the storage stack cannot. However, even threads of executions may not directly correlate with logical tasks. For example, a database server could have a thread pool that it uses to service queries and which exact thread services a query is not fixed. Therefore, in this case, proportional shares cannot to assigned to the threads for achieving application-specific performance insulation.

In this section we present *Context-Aware Scheduler* (CA-*schedule*), a flexible proportional-share disk scheduler that uses application-level logical contexts to identify tasks and insulate performance for the tasks based on their resource shares.

11.1 Design

CA-schedule is a time-slice based proportional-share disk scheduler that uses resource share allocations associated with individual logical contexts, to make scheduling decisions. The share values for each logical context have to be preset by an offline communication channel between the applications and the scheduler. CA-schedule decides the next I/O request to be scheduled, based on the share proportion assigned for the particular logical context and the time each context has consumed. CA-schedule ensures that a particular context is given the proportion of disk-time which is *at least* equal to its share value, provided the context has enough I/O traffic to make use of it. This is irrespective of how many distinct contexts perform I/O simultaneously and the nature of their workloads. For example, if a particular context is assigned a 30% share value, no matter how congested the disk

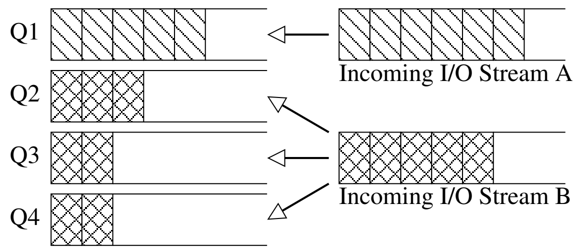


Figure 11.1: *CA-schedule operation when there are two contexts A and B, each with shares 1/4 and 3/4 respectively.*

is, the overall disk-time allocated for that context will at least be 30%.

CA-schedule maintains several request queues based on the granularity of proportions needed. If the minimum granularity of a proportion is $1/n$, then CA-schedule maintains n request queues. CA-schedule uses equal time-slices to service each of these n queues. However, based on the share value of a context, its requests are striped across several queues. For example, if the value of n is 10, and the share level for context A is $1/2$ then the requests from that context are striped across five different queues exclusively. If we assume there are five other contexts each with share level $1/10$, this mechanism will ensure that the requests of context A will be serviced five times for every single time any other context is serviced. The operation of CA-schedule is shown in Figure 11.1.

CA-schedule maintains a table to map context identifiers with their share values. Upon receiving a new request, it looks up the share value for it and adds it to the appropriate queue. Note that it also keeps track of the queue to which the last request for a particular context was added, so that it can determine the queue for the next request from the same context for striping purposes.

This method works for arbitrary values of n which should be decided based on the total number of distinct contexts and the minimum share value that needs to be assigned. Note that CA-schedule only uses operation information and not the data pertaining to that operation. Therefore, it uses access-bound or dual contexts alone. It is useful to have repeatable contexts for CA-schedule, as otherwise, assigning share values have to be done every time a new context identifier is used.

11.2 Implementation

We implemented CA-schedule by modifying the existing fair-share scheduler in the Linux kernel version 2.6.15, popularly called as Complete Fair Queuing (CFQ) scheduler. The CFQ scheduler in Linux already supported time-slice based scheduling at a per-process level based on Linux pid. However, it associated equal shares for all processes. We modified it to perform proportional-share scheduling by having a constant number of queues based on the value of n and to stripe requests on the corresponding queues, based on associated contexts. The modification was quite simple: we modified 40 lines of existing code and added 120 lines of new code to the scheduler.

(a) Sequential Read

	Run 1 (Ops)	Run 2 (Ops)	Run 3 (Ops)
context A (1/8)	Null	193k (144k)	142k (108k)
context B (5/8)	861k (861k)	728k (718k)	540k (538k)
context C (2/8)	Null	Null	298k (215k)

(b) Random Read

	Run 1	Run 2	Run 3
Context A (1/8)	Null	4k (4k)	3k (3k)
Context B (5/8)	21k (21k)	17k (17k)	12k (13k)
Context C (2/8)	Null	Null	6k (5k)

Table 11.1: Read micro-benchmark for CA-schedule: Each column in the table presents the total number of 4KB reads performed in a five minute interval. The values specified in braces is the ideal number of reads that should have been performed based on the share-level for that context. Each row indicates a particular context run in parallel with other contexts in that column. A "Null" value in a column indicates that the process for that context was not run in parallel.

11.3 Evaluation

We evaluated CA-schedule for random and sequential I/O operations. We assumed three different contexts A, B, and C for all tests with share values $1/8$, $5/8$, and $2/8$ respectively. We describe the results for read and write workloads below.

Read Micro-benchmark To evaluate CA-schedule for reads, we ran 10 identical processes for each context. For a sequential read workload each process performed several 4KB sequential reads on their own file. We calculated the total number of reads completed by the threads of each context every minute. We use this as a metric to evaluate the correctness of our proportional share scheduler. We measured the percentage slowdown experienced by each context as other contexts were run in parallel and compared it with each context's share allocation. To get the total capacity of the system, we first ran one context alone and measured the number of reads performed after 5 minutes. We then ran multiple contexts and calculated the slowdown experienced by each of them because of other contexts. We then compared these slowdown percentages with the total capacity of the system. For random read, we used the same number of identical processes and each of them performed random 4KB reads on their own files.

Table 11.1 shows the number of operations completed by each of the contexts when run together with other contexts. Overall the total the slowdown experienced by each context is proportional to the share allocation.

Note that when there are no requests from a particular context, the share allocation for that context will be proportionally distributed among all other active contexts. Therefore, when contexts A and B are alone run without C, even if their individual share values are $1/8$ and $5/8$ respectively, the effective proportion of resources allocated for them would be $1/6$ and $5/6$ respectively. This can be verified based on the values shown in Table 11.1.

(a) Sequential Write

	Run 1 (Ops)	Run 2 (Ops)	Run 3 (Ops)
Context A (1/8)	Null	248k (204k)	200k (153k)
Context B (5/8)	1.2M (1.2M)	938k (1.02M)	536k (765k)
Context C (2/8)	Null	Null	373k (306k)

(b) Random Write

	Run 1	Run 2	Run 3
Context A (1/8)	Null	13k (14k)	10k (11k)
Context B (5/8)	84k (84k)	65k (70k)	46k (53k)
Context C (2/8)	Null	Null	19k (21k)

Table 11.2: Write Micro-Benchmark for CA-Schedule

	Context A (1/6) Response Time (s)	Context B (5/6) Response Time (s)
Transaction	187	109
Delivery	139	80
New Order	131	77
Order Status	135	78
Payment	160	97
Throughput (tpmC)	90.33	139.65

Table 11.3: *Average response time for TPC-C Benchmark*

Write Micro-benchmark To generate sequential and random write workloads, we used 3 threads per context, and each of them performed 4KB write operations on their own files. To introduce synchrony in the workloads, each process performed an `fsync` on their files periodically. We set the sync frequency as 1000 for sequential writes and 50 for random writes. The results for the write workload is shown in Table 11.2. As we can see, the percentage slowdown of each context is in proportion to its share allocation.

The read and write micro-benchmarks detailed above depict heavily parallel workload environments as representative of large scale systems. Hence under most parallel environments, CA-schedule provides accurate resource allocations.

TPC-C Benchmark We ran two instances of the TPC-C benchmark with two different context identifiers, A and B, to evaluate CA-schedule under a relatively less parallel I/O-intensive workload. Contexts A and B had share values of $1/6$ and $5/6$ respectively. Table 11.3 shows the response time and throughput results of the benchmark. As seen from the results, the performance of the contexts are not directly proportional to the share values. This is because, each instance of TPC-C does not generate enough I/O requests with a given time to fill all the queues of CA-schedule. Therefore, our prototype of CA-schedule is more beneficial for workloads that are heavily I/O intensive.

Chapter 12

Related Work

In this chapter, we discuss related research for the concepts, techniques, and insights used in our abstractions and the case-studies that we developed.

12.1 Briding the Information-gap in the Storage Stack

Several systems have been proposed with the overall goal of bridging the information-gap in the system stack. In this section, we classify existing research in this area into four categories: extensible systems, richer abstractions, hint-based interfaces, and inference-based systems. The related work for the case-studies for each abstraction is discussed under their respective sections.

12.1.1 Extensible Systems

Building extensible systems are a solution to the problem of information-gap in the storage stack. Extensible operating systems [10, 65] allow applications to implement their own policies for traditional operating system tasks, by ensuring a safe execution environment them. A related approach is the one taken by Exokernel [20], which advocates building a minimal operating system and have everything else be implemented in application libraries.

The notion of extensibility has also been explored at the hardware level. For example, active disks [1, 62] enable applications to download code into the disk that is run within the disk controller. Such code can implement arbitrary filtering of data based on application level predicates, and even perform more sophisticated operations such as search [45] without actually transferring data out of the disk subsystem. Scriptable RPC [48] proposes making the interface of a network file server extensible so that clients can dynamically implement flexible cache consistency and concurrency policies.

All these systems provide a lot of control to the application and in the process, essentially ties them together. For applications to actually use such extensible layers, they need to have a reasonably intricate understanding of the system, thus making them complex to design. Nevertheless, for applications that really require such control and can utilize it sensibly, these provide the right level of abstraction.

12.1.2 Hint-Based Interfaces

A more evolutionary approach that past research has explored is to provide specific primitives at the system level that the applications can use to convey information to the operating system. Informed prefetching [78] is an example of such a system. By enabling the application to convey information on its future access pattern, the operating system acquires knowledge about the application that it uses to perform more intelligent prefetching. Another example is the Logical disk [17], which provides an interface for the applications to encode locality hints by creating lists of blocks. Researchers have also looked at the flipside of the problem: provide information about the operating system to the application so that the application can make intelligent decisions. Infokernel [5], and icTCP [31] advocate the approach of the operating system exporting a minimal amount of internal information which the applications then use to tune their behavior.

Previous work has also looked at the idea of conveying application knowledge through new abstractions. Perhaps the closest to our work is the idea of Resource Containers [22], which allows applications to group requests into a resource container which is then treated as a logical principal for the purposes of resource isolation. However, even Resource Containers were built with the specific goal of resource accounting.

One commonality between many of these hint-based approaches is that the hints are often tied to a specific kind of optimization or functionality. In other words, the information being transferred is designed with a particular purpose in mind. This in turn limits the flexibility of such a system because each new class of functionality may require yet another new primitive to be added to the interface.

12.1.3 Richer Abstractions

Our work is closely related to a large body of work examining new interfaces between file systems and disk storage. For example, logical disks expand the block-based interface by exposing a list-based mechanism that file systems use to convey grouping between blocks [17]. The Universal File Server [11] has two layers where the lower layer exists in the storage level, thereby conveying directory-file relationships to the storage layer. More recent research has suggested the evolution of the storage interface from the current block-based form to a higher-level abstraction. Object-based Storage Device (OSD) is one example [53]; in OSDs the disk manages variable-sized objects instead of blocks. Object-based disks handle block allocation within an object, but still do not have information on the relationships across objects. Another example is Boxwood [49]; Boxwood considers making distributed file systems easier to develop by providing a distributed storage layer that exports higher-level data structures such as B-Trees. ExRAID [18] explores the utility of exposing hardware specific information from a RAID device to the higher layers such as the file system.

These interfaces are designed with some specific applications or scenarios in mind. For example, it is hard to implement a database in an object-based disk. This illustrates that it is hard to design a generic interface that is suitable for a wide-range of applications.

12.1.4 Inference-Based Systems

Inference-based systems take the extreme approach of making no modifications to interfaces, but *infer* cross-layer information without explicit transfer for information across the layers.. Gray-box systems [4] is an early example of such an approach. An application with "gray-box" knowledge of the operating system attempt to implicitly control the operating system behavior by tuning its workload in such a way that it takes the operating system to a state that results in the desired policy. Another system built along the same philosophy is semantically-smart disks [70] in which the storage system infers knowledge about the higher layers by carefully observing traffic patterns and correlating them to higher level operations.

Although inference-based techniques are valuable from the viewpoint of being easily deployable and less intrusive, these approaches have their own limitations because they are heavily constrained in terms of not changing interfaces. This in many cases results in additional complexity, making it hard to reason about correctness while also limiting the usage of such inferred knowledge to less aggressive applications that can tolerate inaccuracy.

12.2 Interface Between File Systems and Disks

Our work is Type-Safe Disks is closely related to a large body of work examining new interfaces between file systems and storage. For example, logical disks expand the block-based interface by exposing a list-based mechanism that file systems use to convey grouping between blocks [17]. The Universal File Server [11] has two layers where the lower layer exists in the storage level, thereby conveying directory-file relationships to the storage layer. More recent research has suggested the evolution of the storage interface from the current block-based form to a higher-level abstraction. Object-based Storage Device (OSD) is one example [53]; in OSDs the disk manages variable-sized objects instead of blocks. Similar to TSD, object-based disks handle block allocation within an object, but still do not have information on the relationships across objects. Another example is Boxwood [49]; Boxwood considers making distributed file systems easier to develop by providing a distributed storage layer that exports higher-level data structures such as B-Trees. Unlike many of these interfaces, TSD considers backwards compatibility and ease of file system modification as an important goal. By following the block-based interface and augmenting it with minimal hooks, we enable file systems to be more readily portable to this interface, as this paper demonstrates. Others examine the storage interface by trying to keep the interface constant, but move some intelligence into the disk system. For example, the Loge disk controller implemented eager-writing by writing to a block closest to its disk arm [21]. The log-based programmable disk [85] extended this work, adding free-space compaction. These systems, while being easily deployable by not requiring interface change, are quite limited in the functionality they extend to disks.

A more recent example of work on improving storage functionality without changing the interface is Semantically-smart Disk Systems (SDSs) [71]. An SDS enables rich functionality by automatically tracking information about the file system or DBMS using the storage system, by carefully watching updates. However, semantic disks need to be tailored to the specifics of the file system above. In addition, they involve a fair amount of complexity to infer semantic information underneath asynchronous file systems. As the authors point out [69], SDS is valuable when the interface cannot be changed, but serves better as an evolutionary step towards an eventual change to an explicit interface such as TSD.

12.3 Type-safety

The concept of type safety has been widely used in the context of programming languages. Type-safe languages such as Java are known to make programming easier by providing automatic memory management. More importantly, they improve security by restricting memory access to legal data structures. Type-safe languages use a philosophy very similar to our model: a capability to an encompassing data structure implies a capability to all entities enclosed within it. Type-safety has also been explored in the context of building secure operating systems. For example, the SPIN operating system [10] enabled safe kernel-level extensions by constraining them to be written in Modula-3, a type-safe language. Since the extension can only access objects it has explicit access to, it cannot change arbitrary kernel state. More recently, the Singularity operating system [33] used a similar approach, attempting to improve OS robustness and reliability by using type-safe languages and clearly defined interfaces.

12.4 Capability-based Access Control

Network-Attached Secure Disks (NASDs) incorporate capability based access control in the context of distributed authentication using object-based storage [2, 25, 54]. Temporal timeouts in ACCESS are related to caching capabilities during a time interval in OSDs [6]. The notion of using a single capability to access a group of blocks has been explored in previous research [2, 29, 54].

In contrast to their object-level capability enforcement, ACCESS uses implicit path-based capabilities using pointer relationships between blocks.

12.5 Notion of Context in Storage

The idea of tagging requests with identifiers has been explored in the context of distributed systems for performance debugging, profiling, etc. Pinpoint [15] and Magpie [8] are examples of systems in this category. Recently, Thereska et al. proposed applying a similar idea in the context of distributed storage systems mainly for performance monitoring [77]. All these systems look at tagging requests in a causal chain with a certain identifier so that the entire *path* of a logical request (which may involve multiple physical network hops) can be tracked. Researchers have also looked at implicitly inferring this causal knowledge without explicit tagging [3, 28, 46] but it involves significant complexity compared to the explicit tagging approach. These systems only operate within the scope of one logical request and are targeted at a specific application. In contact, CAIO allows for a more general expression of application level semantics to cater to a wide variety of applications.

Previous work has also looked at conveying application-level grouping through new abstractions similar to our notion of context. Perhaps the closest to our work is the idea of Resource Containers [22], which allows applications to group requests into a resource container which is then treated as a logical principal for the purposes of resource isolation and accounting. However, similar to the systems discussed above, resource containers were also built with the specific goal of resource accounting and convey information on one specific kind of grouping.

Our work on context-aware I/O also fits into a class of other work on general solutions for bridging the information gap across system layers. Work in this area mainly belongs in three categories: extensible systems, hint-based interfaces, and implicit techniques to infer information or exert control. We discuss each of these.

12.6 File System Consistency

Consistency mechanisms for file systems have been explored extensively. Early file systems such as FFS [52] relied on a global scan of disk metadata to fix consistency problems. This mechanism, called the file system consistency check (fsck) was in popular use until recently in the Linux Ext2 and Windows VFAT file systems. However, as increasing disk sizes made such global scans more and more expensive, more efficient mechanisms have become popular. Journalling, originally proposed as early as in the Cedar file system [27], uses database like transactions for metadata updates. Modern file systems such as Ext3 and Windows NTFS use journalling for file system consistency. Another technique proposed for file system consistency is Soft Updates [24, 51], which orders updates carefully so that pointer dependencies get updated in the right order. Soft updates is somewhat similar in spirit to our approach since it is also pointer-based. A relatively recent study evaluated the trade-offs between journalling and soft updates [66].

Database systems have for long used mechanisms for consistency. Consistency in databases is enforced via transactions; the ARIES transaction based recovery mechanism [56] is used quite widely in database systems. The basic technique is to group all related updates into a single transaction that is then committed to disk atomically, so that the state remains consistent. As we described in Section 5.6, transactions are more general and powerful than pointer-based consistency, but using transactions requires a fair bit of work at the application level. Our mechanism provides a simpler yet effective alternative to transactions, although not as general.

Consistency at the disk level has been explored in the context of Semantically-smart disks (SDS) [71]. In that paper, the authors implement journalling underneath unmodified Ext2 by utilizing inferred semantic knowledge. However, in their work, the disk system had to be aware of the specific structures at the file system level and thus was tied to a specific file system. Further, it required a synchronous mount of the file system. Our work explores enforcing consistency in a manner generic to the higher level software. However, in the process, we require changing the file system or software above to use the pointer API. We therefore view both these approaches as complementary.

Chapter 13

Conclusions

As Butler Lampson said, interface design is one the most complex aspects of system design, while also being the most important. Interface designers have traditionally embraced the philosophy of minimalism—hide as much information about the layers as possible, so that the layers can innovate and evolve independently. This approach, despite all its merits, has the downside of obscuring what a layer knows about its inputs, thus limiting functionality. At the other extreme, some systems have explored how to completely tie the layers together, by having extensible layers, or exposing detailed information about the inner semantics of a layer. What we have explored in this thesis is a middle-ground, where we send a small amount of information across layers. By making the generation of the information separate from how the information is used, we enable the layers to be independent of each other, while still enabling arbitrary grouping and relationships to be conveyed across the storage stack.

13.1 Lessons Learned

We now discuss four key lessons learnt through our experience in evolving and prototyping our end-to-end abstractions and the case-studies. We believe these lessons would be useful for future interface designers not only in the storage domain, but also more generally in computer systems.

> ***Lesson 1: Generalizing structural and operational information in storage is possible.***

Our pointer abstraction shows that higher-level structures such as files, directories, database tables, or B-trees can be formalized in a generic manner by way of pointers. The fundamental insight behind the pointer abstraction is that today's disk systems store data in the form of fixed size blocks. Therefore, to implement higher-level structures on top of this simple abstraction, relationships have to be established between these individual blocks. Most file systems and other storage software today maintain these relationships through *explicit* pointers. Even if pointers are *implicit* as in the case of extent-based storage design, it is straightforward to generate them explicitly for communicating to the storage stack.

The context-aware storage abstraction provides a means to formalize *operational* information in addition to structural knowledge. By way of hierarchical context identifiers, we show how application-level operational contexts can be encoded in a generic manner even for complex storage applications such as databases.

> ***Lesson 2: Requiring just implementation-level modifications to existing infrastructures is a virtue in interface design.***

Both our abstractions require only implementation-level modifications to existing software layers. Our straightforward implementations of the Ext2TSD and VFATTSD file systems that support the type-aware storage abstractions indicate that as long as there is not a need to redesign existing infrastructures, interface changes are easy to be adopted and deployed. The limited changes that we made to the MySQL and the Linux kernel to support hierarchical contexts corroborate this fact.

> ***Lesson 4: Annotating pointers or contexts with application-level attributes enables a wider range of functionality.***

To support new features that need to be tuned for specific applications or storage layers, annotating generic information with optional *attributes* proves to be useful. Some of our case-studies such as DHIS (Section 6) and CA-schedule (Section 11) use such attributes. These case-studies show that attributes need not be part of the main interface, but can be communicated *offline* between specific layers. For example, the share proportion for different contexts in our CA-schedule proportional-share disk scheduler is set offline by the administrator, specifically in the disk scheduler layer.

> ***Lesson 3: Decoupling the generation of information from its usage has its own limitations***

Although our abstractions enable a wide-range of new functionality in the storage stack, they cannot support certain kinds of features that require precise application-specific information. For example, although type-awareness enables disks to group blocks based on pointers, disks cannot precisely identify if a particular group represents a file, directory, or a database table. Although it is true that a large class of new functionality can be achieved without such such knowledge, some features that needs to use more fine-grained application-specific information cannot be implemented without help from applications. Similarly, although context-aware storage encodes all granularities of application contexts, lower layers cannot identify what each level in the hierarchy means, which may be needed for certain functionality.

13.2 Future Work

In this section, we discuss potential future directions to explore in the topic addressed by this thesis. We first talk about how the general principle behind our abstractions can extend more broadly in other domains. We then discuss two possible future directions to develop new applications using our abstractions.

13.2.1 Generalizing Information in Other Domains

What we have explored in this thesis is how structural and operational knowledge about application data can be formalized and used to bridge the information-gap in the storage stack. This general principle of formalizing the different *properties* of application data is relevant in other domains. For example, it could be interesting to explore if security polices can be formalized in a minimal and generic manner and propagated across the systems stack to enable a new class of secure systems, where different layers can independently provide security features without explicit coordination from applications.

13.2.2 Applications in Virtual Machine Environments

The growing popularity of virtual machine technology exacerbates the problem of information-gap in the systems stack, as it introduces another layer of virtualization. Bridging the gap in this context enables highly useful optimizations and new functionality at the virtual machine host. For example, if the host kernel is aware of structural information about data in virtual machines, it can implement security features such as global anti-virus checking, intrusion detection, or access control, that cannot be bypassed by any guest virtual machine.

13.2.3 Applications in Distributed Environments

Distributed environments present a similar scenario as virtual machines in the aspect of information-gap. We believe that the notion of hierarchical contexts enables a wide range of functionality in distributed systems. Starting from straightforward features such as distributed performance isolation, contexts can potentially go a long way in enabling more complex and interesting functionality such as custom reliability and consistency policies and so on.

13.3 Summary

Overall, we find that type-awareness and context-awareness in the storage stack enables an interesting set of new functionality and optimizations, with minimal modifications to existing infrastructures. We believe that our abstractions explore an interesting and effective design choice in the large spectrum of work on alternative interfaces to storage. As described in Section 13.2, we believe that the insights derived in this thesis apply broadly in several other systems domains.

Bibliography

[1] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 81–91, San Jose, CA, October 1998.

[2] M. K. Aguilera, M. Ji, M. Lillibridge, J. MacCormick, E. Oertli, D. Andersen, M. Burrows, T. Mann, and C. A. Thekkath. Block-level security for network-attached disks. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 159–174, San Francisco, CA, March 2003. USENIX Association.

[3] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 74–89, Bolton Landing, NY, October 2003. ACM SIGOPS.

[4] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and Control in Gray-Box Systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–56, Banff, Canada, October 2001. ACM.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing, NY, October 2003. ACM SIGOPS.

[6] A. Azagury, V. Dreizin, M. Factor, E. Henis, D. Naor, N. Rinetzky, O. Rodeh, J. Satran, A. Tavory, and L. Yerushalmi. Towards an object store. In *Mass Storage Systems and Technologies (MSST)*, 2003.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 164–177, Bolton Landing, NY, October 2003. ACM SIGOPS.

[8] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 2003 ACM Workshop on Hot*

Topics in Operating Systems (HotOS IX), pages 85–90, Lihue, Hawaii, May 2003. USENIX Association.

[9] S. Bauer and N. B. Priyantha. Secure Data Deletion for Linux File Systems. In *Proceedings of the 10th Usenix Security Symposium*, pages 153–164, Washington, DC, August 2001. USENIX Association.

[10] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[11] A. D. Birrell and R. M. Needham. A universal file server. In *IEEE Transactions on Software Engineering*, volume SE-6, pages 450–453, September 1980.

[12] M. Blaze. A cryptographic file system for Unix. In *Proceedings of the first ACM Conference on Computer and Communications Security*, pages 9–16, Fairfax, VA, 1993. ACM.

[13] B. Callaghan, B. Pawlowski, and P. Staubach. NFS Version 3 Protocol Specification. Technical Report RFC 1813, Network Working Group, June 1995.

[14] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[15] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic, internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, pages 595–604, Bethesda, MD, June 2002. IEEE Computer Society.

[16] EMC Corporation. Symmetrix 3000 and 5000 Enterprise Storage Systems. Product description guide, 1999.

[17] W. de Jonge, M. F. Kaashoek, and W. C. Hsieh. The logical disk: A new approach to improving file systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, October 2003. ACM SIGOPS.

[18] T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Bridging the information gap in storage protocol stacks. In *Proceedings of the Annual USENIX Technical Conference*, pages 177–190, Monterey, CA, June 2002. USENIX Association.

[19] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. In *Communications of the ACM*, volume 11, Issue 5, pages 341–346, May 1968.

[20] D. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[21] R. English and A. Stepanov. Loge : A self-organizing disk controller. *HP Labs, Technical Report*, HPL91(179), 1991.

[22] G. Banga and P. Druschel and J. C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 45–58, New Orleans, LA, February 1999. ACM SIGOPS.

[23] G. R. Ganger. Blurring the Line Between OSes and Storage Devices. Technical Report CMU-CS-01-166, CMU, December 2001.

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

[25] G. A. Gibson, D. F. Nagle, K. Amiri, J. Butler, F. W. Chang, H. Gobioff, C. Hardin, E. Riedel, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Langauges and Operating Systems (ASPLOS-VIII)*, pages 92–103, New York, NY, December 1998. ACM.

[26] Michel Gien. Evolution of the CHORUS Open Micro-kernel Architecture. In *Proceedings of the IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 28–30, Cheju Island, Korea, August 1995.

[27] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[28] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 395–408, San Francisco, CA, December 2004. ACM SIGOPS.

[29] H. Gobioff. *Security for a High Performance Commodity Storage Subsystem*. PhD thesis, Carnegie Mellon University, May 1999. citeseer.ist.psu.edu/article/gobioff99security.html.

[30] P. Gutmann. Secure Deletion of Data from Magnetic and Solid-State Memory. In *Proceedings of the Sixth USENIX UNIX Security Symposium*, pages 77–90, San Jose, CA, July 1996. USENIX Association.

[31] H. S. Gunawi and A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Deploying Safe User-Level Network Services with icTCP. In *Proceedings of the 6th Symposium on*

Operating Systems Design and Implementation (OSDI 2004), pages 317–332, San Francisco, CA, December 2004. ACM SIGOPS.

[32] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994.

[33] G. Hunt, J. Laurus, M. Abadi, M. Aiken, P. Barham, M. Fahndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, 2005.

[34] IBM. IBM System Storage DS6800. `http://www-03.ibm.com/systems/storage/disk/ds6000/index.html`, 2007.

[35] IBM. IBM System Storage DS8000 Turbo. `http://www-03.ibm.com/systems/storage/disk/ds8000/index.html`, 2007.

[36] M. Icaza, I. Molnar, and G. Oxman. The linux RAID-1, 4, 5 code. In *LinuxExpo*, Research Triangle Park, NC, April 1997.

[37] Network Appliance Inc. Network Appliance FAS6000 Series. Product Data Sheet, 2006.

[38] InnoDB. Innobase oy. www.innodb.com, 2007.

[39] M. Ji, A. Veitch, and J. Wilkes. Seneca: remote mirroring done write. In *Proceedings of the Annual USENIX Technical Conference*, San Antonio, TX, June 2003. USENIX Association.

[40] N. Joukov and E. Zadok. Adding Secure Deletion to Your Favorite File System. In *Proceedings of the third international IEEE Security In Storage Workshop (SISW 2005)*, pages 63–70, San Francisco, CA, December 2005. IEEE Computer Society.

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

[42] J. Katcher. PostMark: A new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[43] S. King and P. Chen. Backtracking Intrusions. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, October 2003. ACM SIGOPS.

[44] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

[45] L. Huston and R. Sukthankar and R. Wickremesinghe and M. Satyanarayanan and G. R. Ganger and E. Riedel and A. Ailamaki. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 73–86, San Francisco, CA, March/April 2004. USENIX Association.

[46] Z. Li, Z. Chen, S. M. Srinivasan, and Y. Zhou. C-miner: Mining block correlations in storage systems. In *FAST '04: Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pages 173–186, Berkeley, CA, USA, 2004. USENIX Association.

[47] J. Liedtke. On micro-kernel construction. In *Proceedings of the Symposium on Operating Systems Principles*, pages 237–250, Copper Mountain, CO, December 1995.

[48] M. Sivathanu and A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Evolving RPC for active storage. In *Proceedings of the 10th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 264–276, San Jose, CA, October 2002. ACM.

[49] J. MacCormick, N. Murphy, M. Najork, C. Thekkath, and L. Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 105–120, San Francisco, CA, December 2004. ACM SIGOPS.

[50] K. Magoutis, M. Devarakonda, and K. Muniswamy-Reddy. Galapagos: Automatically discovering application-data relationships in networked systems. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*, pages 701–704, Munich, Germany, May 2007. IEEE.

[51] M. K. McKusick and G. R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 1–18, Monterey, CA, JUNE 1999. USENIX Association.

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

[53] M. Mesnier, G. R. Ganger, and E. Riedel. Object based storage. *IEEE Communications Magazine*, 41, August 2003. ieeexplore.ieee.org.

[54] E. Miller, W. Freeman, D. Long, and B. Reed. Strong security for network-attached storage. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST 2002)*, pages 1–13, Monterey, CA, January 2002. USENIX Association.

[55] J. Mogul, l. Brakmo, D. Lowell, D. Subhraveti, and J. Moore. Unveiling the transport, 2003.

[56] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[57] MySQL AB. MySQL: The World's Most Popular Open Source Database. `www.mysql.org`, July 2005.

[58] *Overwrite, Secure Deletion Software*. `www.kyuzz.org/antirez/overwrite`.

[59] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the ACM SIGMOD*, pages 109–116, June 1988.

[60] R. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP '95)*, pages 79–95, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[61] R. Perlman. Secure Deletion of Data. In *Proceedings of the third international IEEE Security In Storage Workshop (SISW 2005)*, San Francisco, CA, December 2005. IEEE Computer Society.

[62] E. Riedel. Active disks: Remote execution for network-attached storage. Technical Report CMU-CS-99-177, Carnegie-Mellon University, November 1999.

[63] M. Rosenblum. *The Design and Implementation of a Log-structured File System*. PhD thesis, Electrical Engineering and Computer Sciences, Computer Science Division, University of California, 1992.

[64] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet small computer systems interface (iSCSI). Technical Report RFC 3720, Network Working Group, April 2004.

[65] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the architecture of the VINO kernel. Technical Report TR-34-94, EECS Department, Harvard University, 1994.

[66] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proc. of the Annual USENIX Technical Conference*, pages 71–84, San Diego, CA, June 2000. USENIX Association.

[67] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[68] G. Sivathanu, S. Sundararaman, and E. Zadok. Type-safe disks. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 15–28, Seattle, WA, November 2006. ACM SIGOPS.

[69] M. Sivathanu, L. N. Bairavasundaram, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Life or death at block-level. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2004)*, pages 379–394, San Francisco, CA, December 2004. ACM SIGOPS.

[70] M. Sivathanu, V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Improving storage system availability with D-GRAID. In *Proceedings of the Third USENIX Conference on File and Storage Technologies (FAST 2004)*, pages 15–30, San Francisco, CA, March/April 2004. USENIX Association.

[71] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-Smart Disk Systems. In *Proceedings of the Second USENIX Conference on File and Storage Technologies (FAST '03)*, pages 73–88, San Francisco, CA, March 2003. USENIX Association.

[72] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE Real-Time Systems Symposium*, December 1996.

[73] M. Stonebraker. Operating System Support for Database Management. *Communications of the ACM*, 24(7):412–418, July 1981.

[74] J. D. Strunk, G. R. Goodson, M. L. Scheinholtz, C. A. N. Soules, and G. R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the 4th Usenix Symposium on Operating System Design and Implementation (OSDI '00)*, pages 165–180, San Diego, CA, October 2000. USENIX Association.

[75] Sun Microsystems. NFS: Network file system protocol specification. Technical Report RFC 1094, Network Working Group, March 1989.

[76] Seagate Technology. Momentus 5400 PSD Hybrid Hard Drives. `http://www.seagate.com/www/en-us/products/laptops/momentus/momentus_5400_psd_hybrid/`, 2007.

[77] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking activity in a distributed storage system. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'06)*, pages 3–14, Saint Malo, France, June 2006. ACM.

[78] A. Tomkins, R. Patterson, and G. Gibson. Informed Multi-Process Prefetching and Caching. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 100–114, Seattle, WA, June 1997. ACM SIGOPS.

[79] Transaction Processing Performance Council. TPC Benchmark C, Standard Specification. `www.tpc.org/tpcc`, 2004.

[80] T. Ts'o. E2fsprogs: Ext2/3/4 filesystem utilities, 2008. `http://e2fsprogs.sourceforge.net`.

[81] S. Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo Conference Proceedings*, May 1998.

[82] VERITAS Software. VERITAS file server edition performance brief: A PostMark 1.11 benchmark comparison. Technical report, Veritas Software Corporation, June 1999. `http://eval.veritas.com/webfiles/docs/fsedition-postmark.pdf`.

[83] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger. Argon: Performance insulation for shared storage servers. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 61–76, San Jose, CA, February 2007. USENIX Association.

[84] C. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[85] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual log based file systems for a programmable disk. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 29–44, New Orleans, LA, February 1999. ACM SIGOPS.

[86] Y. Wang and A. Merchant. Proportional-share scheduling for distributed storage systems. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 47–60, San Jose, CA, February 2007. USENIX Association.

[87] C. Weddle, M. Oldham, J. Qian, A. A. Wang, P. Reiher, and G. Kuenning. PARAID: A gear-shifting power-aware RAID. In *Proceedings of the Fifth USENIX Conference on File and Storage Technologies (FAST '07)*, pages 245–260, San Jose, CA, February 2007. USENIX Association.

[88] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID Hierarchical Storage System. *ACM Transactions on Computer Systems*, 14(1):108–136, February 1996.

[89] C. P. Wright, M. Martino, and E. Zadok. NCryptfs: A secure and convenient cryptographic file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 197–210, San Antonio, TX, June 2003. USENIX Association.

[90] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes. Hibernator: Helping disk arrays sleep through the winter. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 177–190, Brighton, UK, October 2005. ACM Press.