UNIVERSITY OF CALIFORNIA
SANTA CRUZ

PROGRAMMABLE STORAGE

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Noah Watkins

June 2018

The Dissertation of Noah Watkins
is approved:

Professor Carlos Maltzahn, Chair

Professor Scott A. Brandt

Professor Peter Alvaro

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Copyright © by
Noah Watkins
2018

Table of Contents

List of Figures vi

List of Tables xii

Abstract xiii

Dedication xv

Acknowledgments xvi

1 Introduction 1

- 1.1 Dissertation overview . . . 4
- 1.2 Programming data-centric systems . . . 6
  - 1.2.1 Middleware and layering . . . 7
  - 1.2.2 Specialized systems . . . 9
  - 1.2.3 Unified storage . . . 10
  - 1.2.4 Programmable storage . . . 12
- 1.3 Motivation . . . 13
  - 1.3.1 Reliability . . . 13
  - 1.3.2 Hardware trends . . . 14
  - 1.3.3 Data movement . . . 16
  - 1.3.4 Open-source software . . . 19
- 1.4 Methodology . . . 21
- 1.5 Contributions . . . 23

2 Programmable storage 26

- 2.1 Overview . . . 27
- 2.2 Building a storage service . . . 28
  - 2.2.1 Shared-log storage service . . . 29
  - 2.2.2 The CORFU protocol . . . 30
  - 2.2.3 Implementation strategies . . . 33
  - 2.2.4 Programmability approach . . . 37

2.3 Design space challenges . . . 40
2.3.1 Performance portability . . . 40
2.3.2 Complexity . . . 44
2.4 Declarative storage . . . 47
2.4.1 The Bloom language . . . 48
2.4.2 Declarative CORFU protocol . . . 49
2.4.3 Sequencer specification . . . 53
2.4.4 Offline optimization . . . 54
2.5 Conclusion and scope . . . 55

3 Related Work **57**
3.1 Active storage . . . 57
3.1.1 Object-based storage . . . 58
3.1.2 Domain-specific interfaces . . . 59
3.1.3 Parallel file systems . . . 61
3.1.4 Performance management . . . 62
3.2 Storage system extensibility . . . 63
3.3 Software-defined storage . . . 64

4 Data interfaces **67**
4.1 Overview . . . 68
4.2 Programmability . . . 69
4.2.1 Ceph . . . 69
4.2.2 Other storage systems . . . 71
4.3 Physical design . . . 73
4.4 Transactional data management . . . 76
4.4.1 The CORFU abstraction . . . 77
4.4.2 Example design process . . . 78
4.4.3 Summary . . . 93
4.4.4 Declarative approaches . . . 93
4.5 Computational resources . . . 94
4.5.1 Background . . . 95
4.5.2 Skyhook . . . 97
4.5.3 Results . . . 100
4.5.4 Summary . . . 107
4.5.5 Declarative approaches . . . 107
4.6 Structured data management . . . 109
4.6.1 Checkpoint-restart . . . 109
4.6.2 Summary . . . 115
4.7 Durability . . . 115
4.7.1 Persistence . . . 117
4.7.2 Availability . . . 121

4.7.3 Summary . . . 125
4.7.4 Declarative approaches . . . 125
4.8 Graph processing . . . 126
4.8.1 Motivation: log-structured database . . . 130
4.8.2 Programmable storage approach . . . 133
4.8.3 Summary . . . 135
4.8.4 Declarative approaches . . . 135
4.9 Conclusion . . . 136

5 Metadata management **139**
5.1 Overview . . . 141
5.2 File type interface . . . 142
5.3 Name management . . . 143
5.4 Scalability . . . 145
5.4.1 Example: locality . . . 147
5.4.2 Example: scientific data . . . 148
5.5 Shared resources . . . 154
5.5.1 Example: CORFU sequencer . . . 155
5.6 Service metadata . . . 161
5.6.1 Interface propagation . . . 163
5.7 Declarative approaches . . . 165
5.8 Conclusion . . . 166

6 Development environment **169**
6.1 Overview . . . 170
6.2 Motivation . . . 171
6.2.1 Storage interface evolution . . . 173
6.3 Storage development workspace . . . 175
6.3.1 Workspaces . . . 175
6.3.2 Workspace management . . . 177
6.4 The IDE Service . . . 178
6.5 Summary . . . 179

7 Conclusion **181**
7.1 Future work . . . 181
7.1.1 Programmable components and applications . . . 181
7.1.2 Additional systems . . . 182
7.1.3 Declarative storage . . . 183
7.2 Summary . . . 183

Bibliography **185**

List of Figures

1.1 Application and storage stack that is common today. A storage system is composed of many internal services and exposes a fixed set of I/O interfaces. Applications in turn use middleware to map their semantics onto these interfaces. Applications utilize external services when storage system interfaces are not sufficient. 5
1.2 Illustration of a programmable storage system. Internal sub-systems are generalized and reused to create new storage interfaces that otherwise may require external services. 6
1.3 The time required to scan a 140 GB relational database table depending on local processing in which data is move to the client (client-X), and remotely on the storage server (server-X). Scan order is important because it affects locality of data within the cache hierarchy. 17
1.4 RADOS object class usage growth. A class is a functional grouping, and a method represents a specific interface for interacting with objects. 20
1.5 Structure of a programmable storage system. At the top are application-specific interfaces and services. At the bottom are internal subsystems, and the middle represents generalizations of these subsystems. Top-level interfaces are built by mapping semantics onto the generalized services. 22

2.1 The shared-log abstraction presents a totally ordered set of entries addressable by their position in the log. Clients may read and append to the log in parallel. The physical storage for a log is abstracted away and may be implemented in many different ways. 29
2.2 The high-level architecture of CORFU in which high-performance network counters called sequencers assign globally ordered log positions to clients that may then dispatch appends using parallel I/O. 30
2.3 The CORFU device interface is unique. In addition to including an immutable write-once interface, log positions can be invalidated or trimmed for garbage collection, and the maximum position can be queried. 32
2.4 Building storage services as middleware requires introducing external services such as consensus, and implementing complex data management tasks like indexing. 35
2.5 By moving key data management responsibilities into the storage device and exposing an application-specific interface, several challenges are solved that existed in a middleware solution. 36
2.6 Nodes in a distributed storage system typically expose a block-based or object-based interface, and contain numerous sub-systems used to manage data on locally attached storage media. 38
2.7 Four different strategies for implementing the CORFU interface are considered along two dimensions: low-level I/O interface and high-level log partitioning strategy. 42
2.8 Append performance of four shared-log implementations on two versions of Ceph before and after a major release upgrade using the same hardware. 43
2.9 New interfaces are installed at longer timescale than that of individual I/O operations in a system and can correspond to well-defined system life cycle events such as hardware or software upgrades. . . 54

4.1 CORFU is mapped onto Ceph by making use of object interfaces that use programmability to reproduce the requirements of storage devices as defined by the CORFU protocol (e.g. positional r/w, trim, etc...). 78
4.2 The throughput in operations per second of writing 128 byte entries into each of the three storage interface provided natively by Ceph. 83
4.3 The throughput in operations per second of writing 1KB entries into each of the three storage interface provided natively by Ceph. As the size of the entry being stored increases, the variance in performance of the omap interface also increases. 84
4.4 The throughput in operations per second for storing log entries of varying size in either the bytestream or omap interfaces. 85
4.5 The throughput of writing log entries using three strategies: omap, bytestream, and a hybrid in which log metadata is stored in omap, and log entry data is stored in the bytestream. 89
4.6 Simulated worst-case serialized bitmap size with a random arrival order of bits being set. 10K bits. 91
4.7 Simulated partial arrival order. Arrivals are out of order, but bounded by a sliding window across the total set of log positions. 92
4.8 (a) The scalability of single-node database systems are physically limited. (b) Common scale-out approaches have operational limitations that make scalability inflexible due to storage affinity and static resource allocation. 96
4.9 Skyhook stores relational tables by partitioning the table into shards and storing each shard in an object within the Ceph storage system. This allows transparent rebalancing to Skyhook. 98

4.10 Skyhook is structured as a standard single-node database connected to a scale-out storage cluster such as Ceph. The database node connects to each storage server through a custom database-specific interface that allows the database to use storage system resources such as the CPU, and indexing services to accelerate query performance. 99
4.11 Runtime of a 10% selectivity query executed on a single client compared to using custom object interfaces and push-down predicate evaluation. 101
4.12 Runtime of a 10% selectivity regular expression query executed on a single client compared to using custom object interfaces and push-down predicate evaluation. 103
4.13 The runtime of evaluating a point query over 10,000 objects by scanning at the client, scanning at the server, and using an index over all rows. 104
4.14 The runtime cost of storing and indexing table partitions as a function of the number of partitions. 106
4.15 The PLFS abstraction hierarchy. The top-level file view maps writes to per-process logs that are automatically indexed by low-level active objects. 111
4.16 Data interfaces used by PLFS file abstraction. An index is automatically generated, and logical views can be constructed. 112
4.17 Index compression pipeline. 113
4.18 Index compression ratios for 82 PLFS traces. 114
4.19 A sequencer service implemented as a data interface. Sequencer service failover is achieved by using the availability features found in the storage system. Above we have configured the service with 2 replicas such that after 2 failures the service is still functional. . 120
4.20 High-level architecture of Hyder in which clients append deltas to a shared-log, encoding the effects of a transaction. Uncached nodes are paged-in from log storage. 130

4.21 The number of log entries read to access any node in a test database consisting of 10K entries, under a variety of scenarios. 131
4.22 The factor increase in total nodes read over nodes required to satisfy a point query for any key in a database. 132

5.1 On the left is an example organization of file metadata in which a Skyhook database names individual tables. On the right log0 represents a CORFU log and log1 represents a CORFU log with sub-streams. Each of these examples demonstrates how the file namespace can be used to organize application-specific metadata. 144
5.2 MapReduce processes logical partitions in map tasks and matches each map task with physical locations to form an execution plan. The line labeled L is a contribution of SciHadoop which utilizes physical layout knowledge during partitioning 148
5.3 Legion data model and mapping of an adaptive mesh refinement application data to distributed and hierarchical storage. 152
5.4 Each dot is an individual request, spread randomly along the y axis. The default behavior is unpredictable, “delay" lets clients hold the lease longer, and “quota" gives clients the lease for a number of operations. 157
5.5 Throughput of best-effort and 100K quota policies. The best-effort policy spends more time communicating the capability between clients. 158
5.6 The latency of acquiring new log positions using a best-effort and quota-based capability policy. Latency is generally lower with the best-effort policy because the system attempts to balance the capability across clients based on time. 159
5.7 Cluster-wide interface update latency, excluding the Paxos proposal cost for committing the service metadata interface. 163

6.1 Log data is stored in objects that are batch analyzed while developers create new features and evolve the system. 172

6.2 Developers evolve application software and storage interfaces through a co-design process. 174
6.3 In (a) clients use an IDE service to create workspaces that form a context within the storage system. In (b) base data is not duplicated, and CoW provides isolation for interface private data. . . . 176

List of Tables

1.1 A variety of object storage classes exist to expose interfaces to applications. # is the number of methods that implement these categories. 20

4.1 Application-specific log access methods, and a standard log *read* interface. 134

Abstract

Programmable storage

by

Noah Watkins

Storage system solutions have historically been dominated by proprietary offerings designed around a fixed set of common interfaces such as the POSIX file abstraction. However, as the scalability requirements of applications have grown, these interfaces and their implementations have not kept pace. This has forced developers to rely on middleware and external services to address these limitations. These solutions introduce new complexity into the system in the form of duplicated software and can lead to increased costs and reduced reliability. However, the recent availability of high-performance open-source storage systems is allowing developers to explore alternative storage interfaces that directly meet the needs of applications without the fear of vendor lock-in.

We introduce programmable storage as a means by which existing internal storage system abstractions can be generalized and reused to support applications through the creation of domain-specific interfaces. By reusing internal, code-hardened sub-systems applications can avoid duplicating complex software and increase reliability, as well as realize application-specific optimizations. We demonstrate programmable storage by mapping a wide range of common application and storage services requirements onto existing abstractions found in distributed storage systems.

We show that programmable storage introduces real challenges for issues of portability and maintenance, and that the design space for new storage interfaces is intractable for non-expert developers. To address this limitation we propose

that new storage interfaces and services be expressed using a declarative language that abstracts across the differences in internal storage system interfaces to allow application developers to create new storage services without becoming storage system experts. While a declarative approach to building storage interfaces resolves many issues, it doesn't address the implications of developing and evolving storage interfaces in a real-world system. To address this we propose a set of abstractions for developing new interfaces that are aligned with existing software development workflows and source code control systems.

To my family for providing the opportunity to explore.

And to Douglas Niehaus who started me on this journey.

Acknowledgments

First I want to thank my advisor Carlos Maltzahn. His patience, knowledge, and encouragement have been invaluable. I also want to thank my committee members Scott Brandt and Peter Alvaro who have provided great advice and feedback over the years. Santa Cruz is full of people I admire, and I'm lucky to have been able to work with some of them: Neoklis Polyzotis, Shel Finklestein, and Ike Nassi.

One upside to spending so much time in graduate school is that I've had a lot of opportunities to work at different organizations and collaborate with a wide variety of people. I'm grateful to many for making this happen year after year, including Pat McCormick, Galen Shipman, John Bent, Gary Grider, Adam Manzanares, Kleoni Ioannidou, Jay Lofstead, Sage Weil, Anna Povzner, Greg Farnum, and Jeff LeFevre.

So many fine souls have passed through the lab, and they've all helped keep me motivated, focused, and provided support in all its many forms. I'm especially indebted to Michael Sevilla, Ivo Jimenez, Dimitris Skourtis, Joe Buck, and Adam Crume.

I want to thank my partner Tanmayi Sai who has always been there for me through this process, and has been so damn patient on all those weekends that I wanted to work. And I definitely could not have gotten through this without my good friends Ben and Addie Spanbock, Alex Beloi, Chris Ciccolella, Enela Pema, and of course those that were there to witness the beginning: Amik Ahamd, Charley Chan, Sara Oberheide, and Andrew Boie.

Chapter 1

Introduction

By all accounts the standardization of storage interfaces such as POSIX file I/O, and a variety of block-based interfaces, among others, has been a major success. For instance, the familiar and intuitive hierarchical data model of file systems has been adapted to serve nearly every type of user across many domains from desktop applications, to enterprise businesses and all forms of science. In many ways survivor bias is at play: the adoption of these standards by operating systems, and throughout the storage industry, has allowed application and system developers to avoid vendor lock-in, and encouraged independent innovation across common interfaces. These standards continue to successfully serve the vast majority of use cases. However, success in domains such as high-performance computing (HPC) is a direct function of scalability, and HPC applications have pushed parallel file systems to their limit. And high-performance computing is not the only driver of scalability. Cloud-based infrastructure has demanded a level of scalability from the onset that precluded traditional models of storage such as file systems from being the primary storage abstraction. As a result, the storage community has seen new storage systems and interfaces being designed and built to accommodate new sets of scalability requirements.

Until recently, the availability of high-performance storage systems that can serve the HPC and cloud communities has been restricted to expensive, proprietary solutions. But within the last decade, access to these types of storage technologies have been democratized by open-source systems like Hadoop and Spark that provide the tools for large-scale data management and storage [121, 124]. Unfortunately, systems like Hadoop serve a narrow use case. Notably absent from the open-source ecosystem have been high-performance general purpose storage systems offering the standard interfaces of the day: POSIX files and block devices for virtual machine hosting. Systems providing these interfaces have remained largely proprietary. The period we are in now is an interesting time for storage systems, because a choice exists between proprietary systems, and open-source alternatives. Developers and companies have taken notice by modifying open systems to accommodate their application-specific needs, rather than pouring massive resources into constructing custom solutions or relying on workarounds to limitations encountered in proprietary systems. The result has been the rapid evolution of storage interfaces towards domain-specific access methods—a new trend, that in its current form, may become unmanageable.

There are two primary methods by which domain-specific storage interfaces are created. The first is through layering, often in the form of reusable middleware, that is used to transform one generic interface (e.g. POSIX files) into an application-specific storage interface (e.g. multi-dimensional array). In many instances middleware provides a natural way for applications in a particular domain to manage data, and it is common for middleware to exploit the semantics of the exposed interface to implement optimizations on top of the underlying storage system. For instance, collective I/O used in HPC systems transforms application-level workloads into I/O request patterns optimized for an underlying parallel

file system [108, 22, 78]. However, the effectiveness of a middleware approach is limited fundamentally by the underlying storage system. And in the case of proprietary systems, an inability for organizations to adapt the black-box system to their needs means that even large expenditures may not be able to provide the needed level of performance or set of features. As an alternative to an approach based on layering or middleware, an increase in the number of purpose-built storage systems characterizes recent developments in which a specialized system is constructed for a particular class of application. For instance distributed object, key-value, and document-based storage systems have become increasingly common in cloud-based environments for their ability to scale [73, 40, 19]. Such systems expose a narrow, but highly optimized interface. But these systems are expensive to build, require their own maintenance and hardware, and can be slow to arrive due to the maturity required in order to establish trust in a critical component like storage.

In contrast to these two approaches, a trend that has recently emerged is the so-called *unified storage* architecture which seeks to generalize storage systems, and expose from a single system, interfaces for a variety of different application domains. This model can be observed in many proprietary systems that provide both enterprise file-based storage, as well as block-based storage for private cloud virtual machines. While unified storage systems offer many benefits such as consolidation of resources, they typically provide a fixed set of interfaces that cover the widest group of users. With the emergence of high-performance, highly modifiable open-source systems, the unified storage system concept is being pushed further. Not tied to an immutable set of interfaces, open-source systems are being modified to support co-designed interfaces on an application-by-application basis without the fear of vendor lock-in. For instance, the HPC Fast Forward project

developed a set of array-based interfaces exposed directly from the underlying storage system designed specifically for HPC applications storing simulation data [10]. Today this co-design approach is taking the form of an ad hoc process in large part because of the narrow optimizations and monolithic designs—typical of storage systems—that make extensibility difficult. As a result, new interfaces are often hard-coded into a system, and optimizations make assumptions about the current system behavior, resulting in performance tuning and portability challenges. Despite the apparent power of co-designed interfaces, this is a largely unexplored paradigm in production storage system development that has for decades enjoyed a static, standardized interface to rally around.

This thesis explores the benefits and challenges that will emerge as co-designing applications and storage interfaces becomes commonplace. While such systems offer obvious benefits such as highly optimized interfaces and cost savings through consolidation, large unsolved problems immediately present themselves. First, increasing the number of interfaces and optimization strategies in a single system introduces additional complexity into already code-hardened critical systems that raise safety and correctness issues. Second, recombining proven services found within a storage system to provide new functionality is error-prone process that is difficult to reason about. Third, failure to bound complexity on critical paths will reduce the amount of performance that can be extracted from new, low-latency devices. And finally, a variety of new interfaces sharing a single system create quality-of-service concerns. Yikes.

1.1 Dissertation overview

The remainder of this chapter provides a detailed overview of the thesis, starting with Section 1.2 that discusses the challenges of programming and building

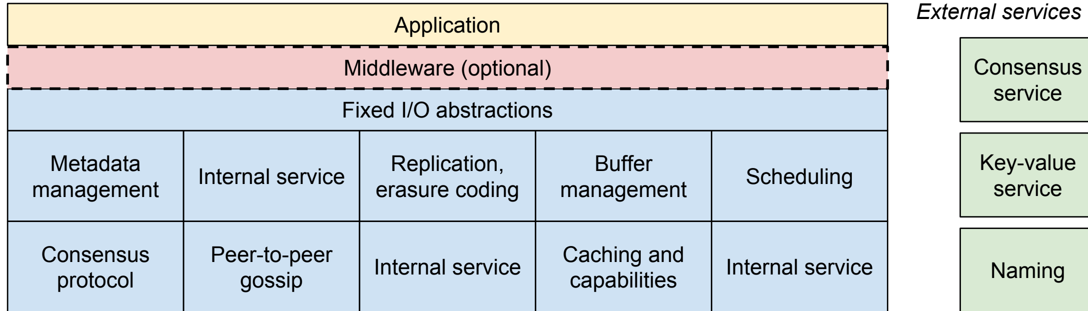


Figure 1.1: Application and storage stack that is common today. A storage system is composed of many internal services and exposes a fixed set of I/O interfaces. Applications in turn use middleware to map their semantics onto these interfaces. Applications utilize external services when storage system interfaces are not sufficient.

storage system interfaces today. Figure 1.1 illustrates the current state of systems by showing the common architecture of application and storage system software stacks today. In this architecture applications map their semantics onto a fixed set of interfaces exposed by a storage system using solutions like middleware libraries. When an application finds that the interfaces exposed by the storage system are not sufficient, external services may be used even if they are services that are otherwise found internally to the storage system.

Section 1.3 describes industry trends that motivate this thesis, and explores the motivation of programmable storage. Figure 1.2 shows an architectural diagram of the programmable storage system we investigate in this thesis. As shown, a programmable storage system exposes generic versions of internal services and supports domain-specific interfaces instead of forcing applications to use a fixed set of abstractions. Creating domain-specific interfaces can simplify applications and improve performance, as we will see throughout this thesis.

Chapter 2 provides an in-depth look at programmable storage, and shows that despite its benefits, the technique can become intractable for developers to

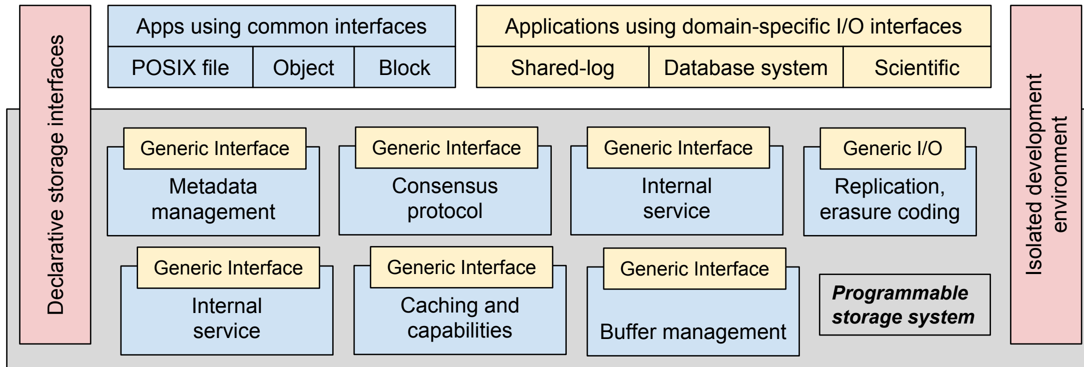


Figure 1.2: Illustration of a programmable storage system. Internal sub-systems are generalized and reused to create new storage interfaces that otherwise may require external services.

manage. As a result, we propose that declarative specifications be used to build new storage interfaces that allow the complex optimization space introduced by programmable storage to be managed by existing techniques found in database systems.

Chapter 4 and Chapter 5 provide an in-depth examination of how programmability interacts with data interfaces and metadata management, respectively. The sub-systems explored in these two chapters provide a substrate to build a broad variety of applications. Finally, Chapter 6 explores the real-world challenges of interacting with a programmable storage system and proposes that the development of new storage interfaces should be tightly integrated into and managed by the storage system.

1.2 Programming data-centric systems

This section provides an overview of the state of programming data-centric systems, especially the primary methods by which new system interfaces are designed and built. We use a general interpretation of the term *data-centric* to

encompass any system, service, or application that is primarily concerned with the storage, management, or processing of data. This covers many categories such as relational database management systems, local and parallel file systems, and embedded and distributed key-value stores. There is a large and active community of research related to data management on locally attached devices that provide low-level storage interfaces such as block-based I/O that includes local files systems such as XFS, EXT-4, and embedded local database such as RocksDB. The interfaces exposed by local data management systems often form the basis for high-level distributed systems which expose a wide variety of interfaces for data management. In this thesis we are primarily concerned with the storage services provided by distributed systems and consumed by scalable applications.

We describe three high-level strategies for developing new storage interfaces that are common today: layering with middleware, purpose-built systems, and unified storage. Each of these approaches have distinct characteristics, though they are not mutually exclusive. After we have detailed these three approaches including some of their shortcomings, we introduce the concept of *storage system programmability*—the focus of this thesis—a fourth approach to developing interfaces based on system extensibility.

1.2.1 Middleware and layering

The first technique is characterized by layers of abstraction implemented on top of existing systems. This is a very common and important type of approach. Existing storage systems such as POSIX file systems provide a robust and hardened substrate for data storage and management. However, applications often store domain-specific, structured data, forcing applications to marshal their data models into and out of a system that provides only management of opaque byte

streams and hierarchical namespaces. Middleware addresses this by providing common interposition layers exposing access methods aligned to a particular problem domain.

Middleware layers are used for more than just convenience. They are frequently used to hide details about an underlying system that affect performance. For instance, MPI-IO is an I/O middleware layer for HPC applications that is used to transform I/O from many independent processes into an I/O workload that is tuned for an underlying parallel file system [41]. This technique is called collective I/O, and is used to aggregate application requests to form large, sequential I/O, potentially using storage system specific tuning parameters such as low-level information like physical data alignment. Unfortunately the POSIX file system standard does not include any concept of alignment or system-specific tuning guidance. Instead, every system exposes its own knobs and tuning parameters through non-standard interfaces to which applications and middleware must adapt.

The dependence on maintaining existing low-level interfaces and the length to which middleware will go can be surprising. Finding the best tuning parameter values for a combination of application, workload, and storage system can be quite difficult. For example, machine learning has been applied to I/O stacks in HPC in order to automatically navigate and tune the configuration space [21]. The parallel log-structured file system (PLFS) was designed to abstract across the individual tuning parameters that each low-level storage system provided, and turn all writes into sequential I/O that tends to perform universally well [22]. The I/O transformation in PLFS uses a per-process log file that accepts each write. In affect, PLFS middleware transparently turns one POSIX file I/O workload generated by an application, into a sharded workload on a different POSIX file

system with equivalent semantics. However, in exchange low-level metadata services become stressed, large indexes are required for accessing data, the file system namespace becomes polluted, and applications cannot access data without using the PLFS middleware [83]. While the performance benefits achieved with PLFS are impressive for many workloads, the drawbacks go beyond simple issues like namespace pollution. The introduction of additional software layers in an I/O stack becomes an issue as storage media and networks become more performant, shifting bottlenecks into the CPU and software itself.

It is also common for distributed applications to require services, such as distributed coordination. In the case of coordination, the POSIX I/O standard provides only coarse-grained support for file locking, and the scalability of these mechanisms will depend on their implementation in the underlying system. As a result, developers may combine an I/O middleware layer with additional systems that provide services such as distributed locking, or build custom solutions and workarounds. This results in added system complexity, maintenance costs, and challenges related to correctly composing services [14].

1.2.2 Specialized systems

Rather than introducing middleware abstraction layers, the bottom-up approach to constructing new storage interfaces is to design and implement entirely new systems. Special-purpose systems can offer a wide variety of data models, optimize for different types of access patterns, and typically address a narrow use case or class of application. Building storage stacks from the ground up for a specialized use case can result in the best performance. For example, GFS [50] and HDFS [101] were designed specifically to serve MapReduce and Hadoop jobs, and use techniques like exposing data locality and relaxing POSIX constraints to

achieve application-specific I/O optimizations. Another example is Boxwood [81], which experimented with B-trees and chunk stores as storage abstractions to simplify building applications. Examples of specialized storage systems can be nearly everywhere, and they cover a variety of data models like graphs, documents, and key-value stores. Many of these systems even offer trade-offs by providing relaxed or eventual consistency semantics, and may even expose knobs that affect durability and availability in an effort to provide more control to applications. For example, both Google and Amazon provide cheaper storage at reduced levels of redundancy allowing applications to select the best policy for each piece of data.

While special-purpose systems can offer targeted performance and scalability improvements, they can introduce inefficiencies in the form of redundancy at multiple levels. This redundancy can include hardware, especially common with storage systems, and may result in underutilized resources when not properly sized for a target workload. Domain expertise is typically required to tune and administer each system. And redundancy can exist in more abstract forms, such as across system internals in the form of services (e.g. consensus services or metadata management). As we will discuss in Section 1.3, these types of redundancy increase costs, and introduce risk as a result of additional system complexity.

1.2.3 Unified storage

Unlike systems that expose a single interface (e.g. POSIX files or special-purpose systems like key-value stores), so-called unified storage systems expose a multitude of storage interfaces from a single system implementation (e.g. a system with POSIX files *and* virtual block devices). And in contrast to approaches to building new interfaces that rely on middleware, interfaces in unified storage systems are natively integrated, allowing them to take advantage of unique opti-

mization opportunities.

Some common examples of unified storage systems are Ceph, IBM Spectrum Scale, and NetApp OnTap which all provide some combination of common interfaces such as shared file systems, or block and object interfaces to support the needs of on-premise cloud deployments. These systems are attractive because they remove redundancy by supporting multiple interfaces in a single system. The result is shared hardware, and software sub-systems, as well as the administrative benefits of a single system to manage.

Unified storage systems are difficult to build. In addition to managing competing workloads, these systems must also take into account the semantics of a variety of interfaces, and the optimization strategies that may benefit these interfaces and the applications using them. When viewed through the lens of the most common set of applications in deployment today—files, block, and objects—unified storage systems would appear to be a near general solution, capable of supporting the vast majority of users. While it is true that these interfaces are used across a large percentage of use cases, as outlined in Sections 1.2.2 and 1.2.1, the demand for a *broad variety* of storage interfaces still exist, as demonstrated by the prevalence of purpose-built systems.

While unified storage systems have demonstrated that a single system can successfully support many different storage interfaces, users rely on vendors and developers with specific storage system expertise to implement new services. Such barriers to entry can lead developers to regress into the aforementioned approaches such as constructing middleware or relying on specialized systems.

1.2.4 Programmable storage

Like unified storage systems, programmable storage seeks to expose many interfaces from a single system, removing redundancy at many levels. In contrast, the aim of programmable storage is to expose existing sub-systems as reusable components that can be used by developers without deep storage system expertise, to construct new interfaces specially designed for a particular application domain.

For example, consider a distributed application that manages data consistency by defining versioned views of a data set. This application will likely require an I/O interface for storing and retrieving data, as well as a mechanism for managing and versioning views of the data. While a distributed storage system may provide I/O interfaces sufficient for storing this application's data, interfaces for view management such as a consensus service are not usually found alongside standard I/O interfaces. As a result applications depend on external systems to provide specialized services like consensus, such as systems implementing Paxos, Raft, or Zab protocols [75, 86, 68]. Crucially, services such as consensus systems are very common, general building blocks in distributed systems, including distributed storage systems. A programmable storage system may expose many internal services to applications to provide an opportunity to avoid relying on additional systems. This highlights an important distinction between programmable storage and software-defined storage (SDS). Application control over a software-defined storage system is generally restricted to a pre-defined set of tuning and configuration parameters. An SDS system may be customizable to a large number of scenarios, but are limited in the scope of semantics they express. In comparison, programmable storage can be complementary by allowing a storage system to provide application-specific functionality and optimizations that integrate with the space of SDS configurations.

As we will show, the utility of exposing internal storage services is not limited to reducing redundancy, but can also be used to improve performance through low-level optimization strategies, exposing locality, and exploiting storage system resources such as CPU and memory.

1.3 Motivation

Several recent trends have motivated the work in this thesis. First, it is of increasing importance that data be safe, secure, and accessible, given its value across nearly every application domain. Yet, we are seeing rapid innovation in storage systems that have historically required long periods of time before trust in them is established. The rapid innovation is being driven by two factors. First, next-generation hardware has a performance profile that is moving bottlenecks away from storage devices and into software. Second, applications and services with high scalability requirements are relying on new storage systems and interfaces to meet their needs.

Building new storage systems is an expensive, time-consuming, and error-prone process. Fortunately, the availability of high-performance open-source software is allowing new techniques and designs to be tested with a higher frequency, and without starting from scratch. Hardened software components that are commonly duplicated across many distributed systems are being reused to build new services, but it is an open question which services and in what form they should be exposed.

1.3.1 Reliability

Storage systems are unique in that they manage data that is often critical to keep both safe and available. Data durability and accessibility may be critical

to business models, such as archiving family photos or providing secure access to medical records, or for science and national security. For these reasons, it can often take many years—upwards of a decade—before new storage systems can be trusted by some users [54]. This means that developers may face many uphill battles when building new storage interfaces.

Construction of purpose-built systems will inherently duplicate non-trivial services that are critical to correct operation. This duplication can range from high-level services such as a consensus engine for managing cluster metadata, to low-level software components for correctly managing data on storage media. Unfortunately each purpose-built system must go through the same process of hardening their implementation of these services, with the most reliable method for hardening being a large amount of developers interacting with the code, and real-world use and deployment. This creates the concern that construction of more systems will result in an overall decrease in reliability as the total amount of deployment time for the average system decreases.

Programmability seeks to address this concern by exposing existing, hardened storage subsystems in a way that applications may reuse them to build new interfaces without affecting correctness. This is easier for some subsystems compared to others. For instance, even though a system could expose a high-level interface to a service such as an internal implementation of a consensus engine, composing otherwise correct services does not necessarily result in a composition that behaves correctly [14].

1.3.2 Hardware trends

The design of storage systems has predominately been centered around block-based devices such as spinning disks. As a result, both applications and stor-

age systems have commonly optimized for sequential, block-based I/O patterns that perform well with spinning disks. Since these devices generally have high-latency and low-bandwidth performance compared to modern networks, CPUs, and DRAM, the focus on optimizing for I/O patterns has worked well. However, next-generation hardware such as non-volatile memory offer finer grained access, and orders of magnitude better latency and throughput than disks. While this type of hardware offers a general performance benefit, systems designed for slow spinning media can often not fully exploit the available performance. The reason is that bottlenecks have started to shift from storage devices into software where normally minor concerns such as code path length and the number of context switches become a central component in achieving performance [72, 107].

In order to fully exploit the performance of these next-generation media, the entire I/O stack, including access methods, may need to be carefully rethought. For example, peak performance from many NVMe devices depends on user-level I/O and networking, and low-level optimizations like CPU affinity and per-thread, per-cpu data structures. These types of low-level optimizations pose challenges when designing access methods and rich high-level interfaces that may inherently require complex code paths. For instance, a typical system may have a direct path to a physical media location for standard read/write interfaces that is easily optimized for new media, while complex APIs like those used with key-value databases necessarily have more complexity. In addition, storage systems may have arbitrary sets of features like snapshotting, transactions, varied consistency models, not to mention sub-systems controlling fault-tolerance and recovery. Each of these interfaces and system features interact to create complexity, parts of which can be found in critical I/O paths. And because this complexity can be difficult to manage, many systems have been designed with fixed I/O paths that handle all

requests no matter the system configuration or interface being used. Thus, it may be difficult to exploit the semantics of an interface or configuration to perform optimizations without significant design disruption.

There are two aspects to adapting storage systems to address challenges such as integrating next-generation media. The first challenge is determining the set of interfaces and services that storage systems should expose, as well as how that is achieved. The second aspect arises directly from the evolution of storage interfaces, and involves the adaptation of applications to use new interfaces. This thesis is primarily concerned with the first aspect, but we are driven by observing trends in application development in which applications are taking an increased role in data management in exchange for access to more flexible, performant, or scalable interfaces.

1.3.3 Data movement

Systems such as Hadoop MapReduce attempt to schedule computation locally to where input data is stored [39]. This has historically been an important optimization because the streaming bandwidth of reading data off a number of disks would quickly surpass the total network capacity. However, rapid innovation in the performance and capacity of commodity networks is reducing reliance on locality as an effective optimization in big data applications, and the claim has been made that the data locality optimization is becoming less important [16]. But there are other factors—in addition to network data movement—that make locality an important optimization parameter in the design of storage interfaces and services such as energy consumption [71], and locality within the storage hierarchy, that we discuss next.

One example of data movement in the memory hierarchy is the use of intelli-

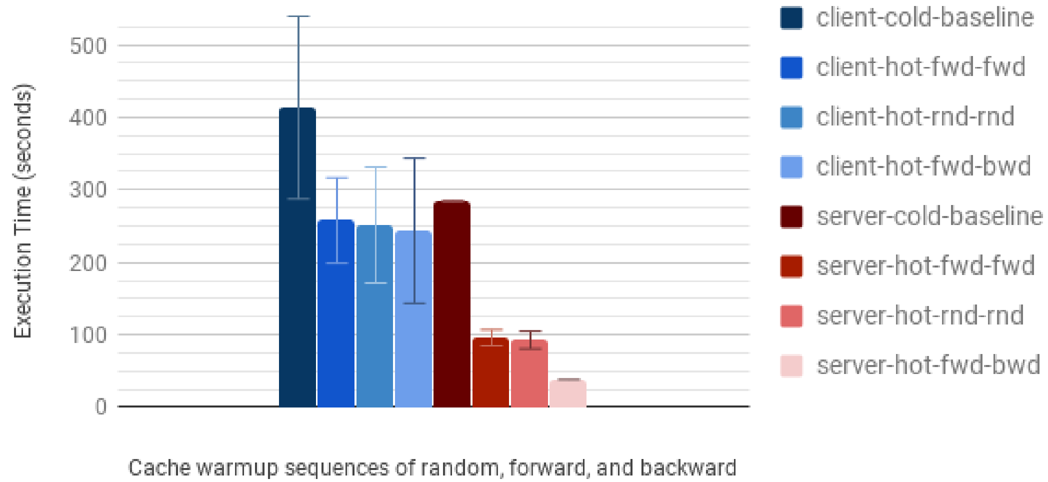


Figure 1.3: The time required to scan a 140 GB relational database table depending on local processing in which data is move to the client (client-X), and remotely on the storage server (server-X). Scan order is important because it affects locality of data within the cache hierarchy.

gent data access methods that help reduce data movement between non-network resources like disks and memory. Take for instance the problem of one or more clients scanning a large data set stored remotely in a distributed object storage system. We assume in this example that the data set is striped across a set of objects, and that the client is insensitive to the order in which the objects are read and processed (e.g. full relational table scan). Given that the storage system must read and return the data requested by the client, it would be advantageous for the client to request objects according to the locality of data in the memory hierarchy of the storage server so as to take advantage data already cached. Unfortunately storage systems do not typically expose this type of information, and applications rely on tracking their own locality of reference to approximate this type of optimization, which increases in difficulty as the number of independent

storage clients increase, and the complexity (e.g. depth) of the storage hierarchy increases.

Figure 1.3 shows the costs of scanning such a data set as a function of the order of objects in the scan. The data set is a 140 GB relational table storing 1 billion rows striped across 10,000 objects. The workload consists of scanning the entire data set looking for a single row. The `client-cold-baseline` and `server-cold-baseline` cases begin from a cold cache state. In each client-`X` case all data is transferred over the network to the client, and in each server-`X` case the data scanning is performed locally on the server storing the data using an application-specific interface discussed further in Section 4.5. The remaining data points differ by the pattern in which the data is scanned. Each scan is performed twice: forward-forward, forward-backward, and random-random. The client-hot-`X` cases perform similarly because the cost of transferring the entire data set dominates. However, performing the scan remotely is affected by the ordering. Since the storage node has slightly less DRAM than the size of the data set, the pattern of I/O will affect the page eviction process when memory pressure is high. Indeed, scanning forward-forward results in a much higher execution time than forward-backward, which is ideal for a basic LRU eviction policy.

Unlike systems such as Hadoop that expose locality information at a high-level, as we have shown some applications can benefit from detailed information like memory hierarchy locality. Other forms of low-level locality that benefit applications include data layout. For instance, multi-dimensional data that is stored in a byte stream can be poorly aligned, making high-level locality information difficult to optimize for. Rather, understanding the low-level layout is beneficial for scheduling I/O [29].

1.3.4 Open-source software

While storage systems and interfaces have primarily evolved in response to market demand, recent trends toward application-specific I/O interfaces is driving out proprietary systems from consideration. Instead, high-performance open-source storage systems are filling this void by providing a platform for innovation without the fear of vendor lock-in. These open systems provide a means for research and development that was difficult to approach in the past. Furthermore, the reusability of software afforded by open systems provides motivation for exploring new system designs as the costs of recombining and reusing existing sub-systems can be far less than building new systems from the ground up.

To demonstrate a recent trend towards more application-specific storage systems we examine the state of programmability in Ceph [118]. Something of a storage Swiss army knife, Ceph simultaneously supports file, block, and object interfaces on a single cluster. Ceph's Reliable Autonomous Distributed Object Storage (RADOS) system is a cluster of object storage daemons that provide Ceph with data durability and integrity using replication, erasure-coding, and scrubbing [119]. Ceph already provides some degree of programmability; the object storage daemons support domain-specific code that can manipulate objects on the server that store the data locally. These "interfaces" are implemented by composing existing low-level storage abstractions that execute atomically. They are written in C++ and are statically loaded into the system. For example, an application might store images in objects and use the CPU resources of the storage system to remotely compress the image before returning it to a user.

The Ceph community provides empirical evidence that developers are already beginning to embrace programmable storage. Figure 1.4 shows a dramatic growth in the production use of domain-specific interfaces in the Ceph community since

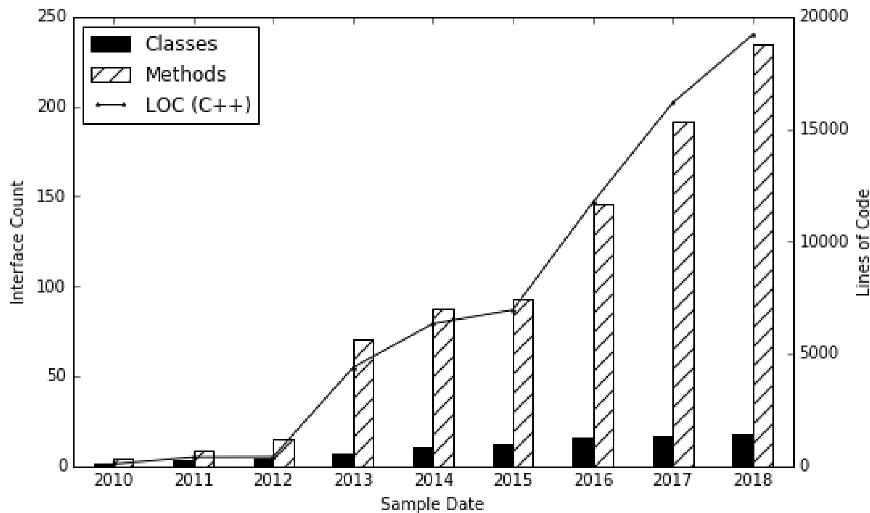


Figure 1.4: RADOS object class usage growth. A class is a functional grouping, and a method represents a specific interface for interacting with objects.

2010. In the figure, *classes* are functional groupings of *methods* on storage objects (e.g. remotely computing and caching the checksum of an object extent). What is most remarkable is that this trend contradicts the notion that API changes are a burden for users. Rather it appears that gaps in existing interfaces are being addressed through ad hoc approaches to programmability. In fact, Table 1.1 categorizes existing interfaces and we clearly see a trend towards reusable services like locking, and metadata management.

Category	Example	#
Logging	Geographically distribute replicas	11
Metadata Management	Snapshots in the block device OR Scan extents for file system repair	74
Locking	Grants clients exclusive access	6
Other	Garbage collection, reference counting	4

Table 1.1: A variety of object storage classes exist to expose interfaces to applications. # is the number of methods that implement these categories.

The takeaway from Figure 1.4 is that programmers are already trying to use programmability because their needs, whether they be related to performance,

availability, consistency, convenience, etc., are not satisfied by the existing default set of interfaces. The popularity of the custom object interface facility of Ceph could be due to a number of reasons, such as the default algorithms/tunables of the storage system being insufficient for the application's performance goals, programmers wanting the ease of programming with application-specific semantics, and/or programmers knowing how to manage resources to improve performance. A solution based on application-specific object interfaces is a way to work around the traditionally rigid storage APIs because custom object interfaces give programmers the ability to tell the storage system about their application: if the application is CPU or I/O bound, if it has locality, if its size has the potential to overload a single node, etc. Programmers often know what the problem is and how to solve it, but until the ability to modify object interfaces, they had no way to express to the storage system how to handle their data.

In general the observation that benefits are found in using programmability extends beyond the programmability of object interfaces in Ceph. Storage systems must control all aspects of consistent metadata management, fault-tolerance, recovery, low-level data management, and networking, among many other subsystems commonly found in today's systems. Each of these components and their behavior is potentially a point of optimization for an application interface.

1.4 Methodology

The approach that we take towards evaluating the feasibility of constructing a programmable storage system occurs along three dimensions. These aspects are illustrated in Figure 1.5. First, we identify high-impact applications and general purpose services that can benefit from programmable storage by eliminating duplication of services or take advantage of domain-specific optimizations. For

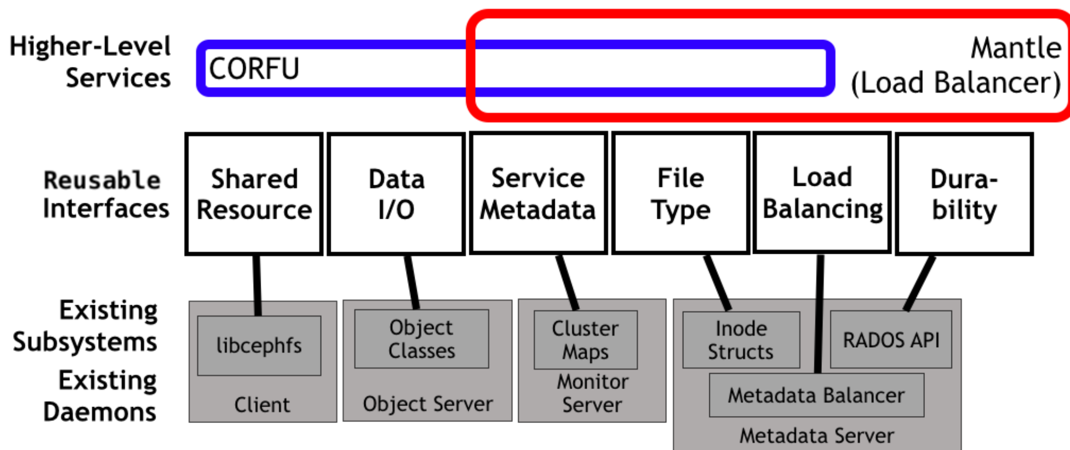


Figure 1.5: Structure of a programmable storage system. At the top are application-specific interfaces and services. At the bottom are internal sub-systems, and the middle represents generalizations of these sub-systems. Top-level interfaces are built by mapping semantics onto the generalized services.

instance, shown in the figure is CORFU a general purpose high-performance log that is a useful storage abstraction for a wide range of applications [18]. Second, we identify existing sub-systems commonly found within a distributed storage system that can be reused to provide a generic service—that is, not tailored to its specific need as simply an internal service. Finally, we identify general classes of services into which each of these reusable components can be placed, and explore how the internal services can be mapped onto the semantics of a generic service. This process results in a set of generic reusable services that are useful for constructing domain-specific interfaces for applications and services.

As we will we present, there are cross-cutting issues identified in this thesis related to the real-world challenges of using a programmable storage system. In particular, challenges related to the complexity of navigating the design space of a programmable storage system is addressed by arguing for the use of declarative programming techniques. Throughout this thesis, when an area of the storage system is identified as a candidate for reusability, we also construct an argument

for how the service and its use by an application may be expressed declaratively. Declarative approaches to building storage interfaces are discussed further in the next chapter.

1.5 Contributions

The first contribution made by this thesis is a definition for programmable storage as a means to facilitate the re-use and extension of existing, internal storage system abstractions to enable the creation of domain-specific services and interfaces. This is important because it reduces duplication of common services that result in resource savings and promotes the use of common, code-hardened systems thereby increasing reliability. Programmable storage is distinct from concepts like active storage, software-defined storage, and in-storage computing. Both active storage and in-storage computing refer to the use of storage system resources (typically the CPU) to execute arbitrary code injected by applications. In active storage and in-storage computing, the system typically lacks any context for injected code and simply acts as a conduit for applications to exploit a hardware resource. In contrast, programmable storage restricts system modifications to the confines of the generalizations of existing code-hardened services and the composition of these services in order to build new application-specific interfaces. Through careful consideration of how services are generalized they can be reused by applications in ways that are orthogonal to correctness, reducing concerns that programmability introduces fragility, a common issue with approaches based purely on code injection. These customizations also go beyond any techniques related to software-defined storage which are typically limited to the scope of configuration and tuning parameters in storage systems with a fixed set of interfaces and services.

Our second contribution is a systematic exploration of the design space of programmability in distributed storage systems for building new interfaces and services. We first examine this in the context of data interfaces and show that a broad variety of application-specific interfaces and services can be constructed programmatically without affecting the safety of the underlying system. We explore this space for applications that manage transactional data, as well as applications that can be accelerated through the use of internal storage system resources like CPU, memory, and I/O bandwidth. We show that more abstract interface properties such as durability can also be programmatically altered to support application-specific optimizations. In support of these efforts we developed an API that allows applications to inject dynamic definitions of storage interfaces using the Lua programming language.

We repeated the process of exploring the design space of programmability for two other major sub-systems commonly found in distributed storage systems: a POSIX file system and a consensus service used internally to manage cluster-level metadata. We introduce the concept of a file type that abstracts naming and metadata management across applications allowing file names to be associated with arbitrary metadata controlled by applications and made available to internal sub-systems in support of programmable customizations. We demonstrate file types by constructing a specialized network service for a high-performance log that depends on an application-specific treatment of an in-memory shared resource. In our prototyping system based on Ceph, we were able to use programmability to reuse the capabilities feature, used to maintain cache coherency across file system clients, to mimic the same semantics required by the distributed log service. And finally we show how a common Paxos based sub-system could be re-used to provide management of interface definitions used by the programmability infrastructure

itself.

The third contribution we make is the introduction of declarative storage, which is the use of declarative specifications in the pursuit of programmable storage system customizations. We show that the power of programmable storage will also be its downfall without a means to make the design space of interfaces tractable; the design space is simply too large for non-expert developers to navigate, and the lack of well-defined internal interfaces results in major challenges supporting portability and even maintaining performance after upgrading system software. We show that existing declarative programming languages can fully specify programmable storage interfaces and that when rooted in formal methods, these languages allow important techniques from the database literature to be applied in the context of storage systems.

Finally we outline a large and diverse set of future research opportunities. While we have demonstrated a broadly useful contribution in programmable storage and shown that declarative specifications can address its major shortcomings, each contribution is accompanied by a broad and deep design space. In each chapter we explore the potential directions for this work along the dimension specific to the chapter topic, namely data interfaces, metadata management, and development processes.

Chapter 2

Programmable storage

In Sections 1.2.4 and 1.3 we introduced and motivated programmable storage as an approach to building new storage services. In review, programmable storage seeks to expose generalizations of internal sub-systems found in distributed storage systems, and defines how these sub-systems can be repurposed to support the creation of new storage interfaces and services. Existing approaches to building new data interfaces tend to rely on middleware and external services. They might depend on the construction of entirely new systems, or introduce fragile changes in a monolithic architecture. Requiring system expertise, and modifying complex mission critical systems will invariably increase short-term and long-term costs. Instead, programmable storage seeks to reduce costs and increase reliability through reuse.

The conceptual approach of reuse advocated by programmable storage to building new services and interfaces is nothing new to anyone who has picked up a book on basic software engineering best practices. Building good abstractions that promote modularity and reuse are common goals of any project producing software. But unlike the challenge of building a new system that is modular from the beginning, a primary goal of programmable storage is to expose exist-

ing sub-systems that are otherwise hidden behind standardized interfaces. This is more than an exercise is software maintenance; existing storage systems contain hardened sub-systems that implement mission critical components protecting data. Leaving these fine-tuned sub-systems unperturbed while allowing new interfaces to be created without sacrificing correctness is an entirely different type of challenge.

As presented, programmable storage may appear to be merely a set of common engineering guidelines, combined with domain-specific motivation, and carefully adapted for an existing environment. Indeed, building a programmable storage system, or adapting an existing system to support programmability, requires applying a large amount of system expertise and software engineering discipline. But as we will see in this chapter, the problem becomes more complex and nuanced as the realities of a large, dynamic storage system are revealed.

2.1 Overview

This chapter fully motivates and details programmable storage as a new paradigm for constructing storage abstractions. We use a distributed shared-log service called CORFU as a driving example and show how storage abstractions such as a log can be mapped onto existing services found within a storage system [18]. The CORFU system is not a system we developed; rather CORFU is a real-world use case storage service that is generally useful, but simple enough to demonstrate the power of programmability. While the importance of the programmable storage approach will become clear, new sets of challenges will also emerge. We will show that even when provided access to generalized versions of some common sub-systems, and armed with significant domain expertise, that an approach to programmability based on low-level abstractions can be intractable. This will be

shown through the use of a motivating example in which the design space for constructing a new storage service is too large to explore without the assistance of optimization techniques commonly found in database systems.

To address this challenge that emerges with programmable storage, the second half of this chapter advocates for the use of declarative specifications of storage interfaces. A declarative specification can act as a bridge between the expression of new storage services and their implementation in a given system. Crucially they can be used to hide low-level details that may differ across storage systems as well as across versions of the same system, and thus reduce the complexity of interface development and maintenance. We demonstrate these ideas by presenting an implementation of the distributed shared-log driving example using a declarative language called Bloom.

2.2 Building a storage service

This thesis makes use of many different example storage services to motivate various aspects of programmability. The first driving example that will be used is that of a high-performance distributed shared-log. In this section we introduce the abstraction and its salient properties, and then demonstrate one aspect of programmability by using existing components found in the Ceph distributed storage system to partially replicate the CORFU abstraction. Throughout this thesis the technique of programmability will be applied in expanded contexts, and cover all aspects of CORFU.

The driving examples we have chosen for this thesis, including the shared-log abstraction, have been selected because of their relative simplicity to understand, as well as their overall usefulness as general purpose storage services.

2.2.1 Shared-log storage service

The abstraction of a shared-log is shown in Figure 2.1, and consists of a globally ordered set of immutable entries containing arbitrary data. The entries of a shared-log are addressable by their position in the log, and new entries are appended after the last entry, expanding the address space of the log.

The abstraction is called a shared-log because many clients may be concurrently reading from and appending entries to the log. And as we will see later, the log may be physically stored in many different ways within an underlying storage system. The value of the shared-log abstraction as a storage service is highlighted by its role as a fundamental building block in file systems, distributed systems, as well as in several recent research efforts focused on cloud-based data management [18, 17] and elastic database storage engines [25, 23, 24].

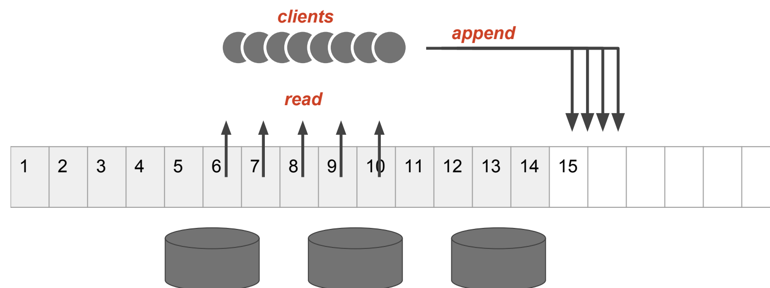


Figure 2.1: The shared-log abstraction presents a totally ordered set of entries addressable by their position in the log. Clients may read and append to the log in parallel. The physical storage for a log is abstracted away and may be implemented in many different ways.

In recent years there has been a resurgence in interest in the shared-log abstraction due to the decreasing cost of storage media that provides high-performance random I/O (e.g. flash memory). This is due to the observation that data management systems that use a log-structured format tend to exhibit workloads with a high amount of random reads which can have significant performance penalties on media such as spinning disk. In the past, this has made a log-structured ap-

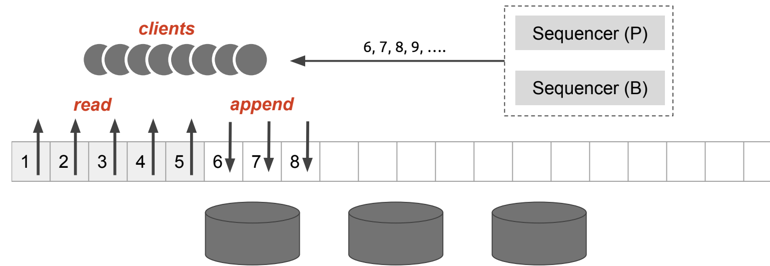


Figure 2.2: The high-level architecture of CORFU in which high-performance network counters called sequencers assign globally ordered log positions to clients that may then dispatch appends using parallel I/O.

proach to certain system designs infeasible, irregardless of the convenience and power of the abstraction.

While the shared-log abstraction is an important building block in distributed systems, typical implementations are often based on consensus protocols (e.g. Paxos [9]) which tend to serialize requests through a primary server. While this has the advantage of simplifying the assignment of a total ordering, scalability becomes a challenging property to achieve. Next we will look at the CORFU protocol which addresses this limitation.

2.2.2 The CORFU protocol

This section introduces CORFU as a driving example for programmability. The CORFU protocol is an approach to building a high-performance shared-log interface [18], and was selected as a driving example because it is a powerful, generally useful real-world system with a protocol that is easy to understand.

The CORFU protocol addresses the bottleneck that some consensus systems have in which writes are funneled through a single node to enforce a global ordering. The issue is addressed in CORFU by decoupling the process of assigning a total ordering to log entries from the actual I/O required to store the data in the log. Such an approach allows clients to complete log I/O directly, avoiding any

centralized I/O proxy that would limit throughput. Note that CORFU is only one way to build a fast shared-log. Indeed there is an active research area that is interested in such designs, and it would be worthwhile to examine other approaches and how their optimization strategies may map onto an existing storage system.

Figure 2.2 shows the high-level architecture of CORFU in which clients read and append to the log concurrently, and a process labeled as *sequencer (P)* assigns a global ordering of log positions to each client. The key insight in CORFU is that the assignment of log positions to clients by the sequencer process can be made very high performance through the use of a volatile network-attached counter (i.e. an in-memory atomic integer), assigning log positions to clients at rates that depend only on the network speed of very small packets, typically at 100K-1M packets per second. The authors of CORFU were able to demonstrate that the sequencer was not the bottleneck in their system, and were able to saturate a large cluster of network-attached flash devices. While a complete description of CORFU is beyond the scope of this thesis, select components will be introduced as necessary as the application of programmability expands to include more aspects of the protocol. For instance, in Figure 2.2 a backup sequencer labeled *sequencer (B)* may take the place of a failed primary. This case is handled by the CORFU protocol, and in Section 4.7 the sequencer and the recovery protocol will be discussed in more detail.

CORFU storage device

In this section we will examine the storage device interface that the CORFU protocol depends on, and show how this interface may be instantiated in an existing system using a variety of techniques with different trade-offs. The application-

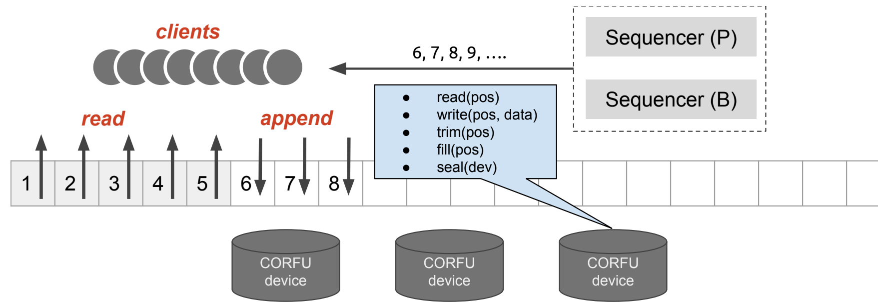


Figure 2.3: The CORFU device interface is unique. In addition to including an immutable write-once interface, log positions can be invalidated or trimmed for garbage collection, and the maximum position can be queried.

specific interface in CORFU makes for an ideal driving example of programmability, but note that a critique of the CORFU interface is beyond the scope of this thesis; we restrict our concern of CORFU to the replication of its salient optimization strategies.

Figure 2.3 depicts the CORFU device interface within the general system architecture. Log clients *read* and *write* entries directly to storage devices using the unique interface shown in the blue box in which a target log position is specified. There are many differences when compared to traditional block-based storage interfaces. First, unlike block interfaces such as SATA or NVMe, the log entry I/O interface in CORFU is write-once, meaning that once a log position has been written, it cannot be written again in the future. And unlike a block device that exposes a fixed set of blocks, each storage device in CORFU is expected to expose a sparse 64-bit address space of log entry positions.

In addition to basic log entry I/O, the interface includes three additional unique components. The *fill* and *trim* interfaces are used to mark a log entry position as being invalid, or free for garbage collection, respectively. And, even after filling or trimming an entry, the log position may not be written or read in the future, further complicating the indexing strategy. Finally, the *seal* inter-

face stores an epoch value in the device and returns the maximum log entry that has been written to. The seal command is used during reconfiguration such as expanding the set of storage devices, or in response to a failure. The use of an epoch is to define a particular view of the system state. Every client request to a storage device is tagged with the epoch value corresponding to the latest view of the system known to the client. The epoch in each request must be at least as large as the epoch value currently stored in the device. The system uses the seal interface to maintain system consistency by coordinating configuration changes to clients and system state without forcing clients to contact a central authority for every I/O request. When a client request is rejected by a storage device for having an out-of-date epoch, the client may refresh its system state from a configuration authority managed by a traditional consensus system such as Paxos.

In the next few sections we will explore methods by which the CORFU device interface may be built or emulated in an existing system by mapping the CORFU interface and semantics onto existing interfaces and services.

2.2.3 Implementation strategies

While the high-level CORFU *protocol* solves a key challenge that storage, application, and distributed system developers may face in achieving high-performance, the storage device specification expected by the protocol is not implemented in any commercially available hardware. However, the CORFU device interface need not be implemented in hardware; software-defined interfaces implemented as middleware or using an RPC proxy are a possibility. We therefore use the CORFU interface as a proxy for application-specific interfaces in our exploration of programmable storage solutions.

In the next few sections three different approaches to building the CORFU

device interface in software will be explored. The first is based on the common approach of using middleware, the second approach places intelligence within the storage device itself, and for the third approach we examine a solution based on storage programmability that addresses limitations in other approaches.

Middleware approaches

One method for building the CORFU interface is to construct it in software on top of standard commercial devices as a layer of middleware. This is a very common approach to dealing with system limitations, or to create new abstractions, and is common in high-performance computing [55, 48]. This architecture is illustrated in Figure 2.4 which depicts the CORFU device interface implemented as middleware between clients and storage devices. In this architecture each client contains an additional piece of software, or accesses the middleware layer remotely as a proxy, that implements the CORFU device interface.

Now consider just one aspect of the requirements of this interface, namely the write-once semantics that make log entries immutable once written. Since the standard interface for devices such as SATA will always allow a write to proceed, and do not provide primitives such as compare-and-swap useful for constructing concurrency control mechanisms, some state must be maintained that records what positions have been written, and enforce the write-once semantics at a higher level, above the storage device. This effectively means that every log entry I/O must access and update authoritative metadata for that log entry. In practice this will likely look like running a round of consensus for every log entry access, or contacting an external service managing log entry metadata. The presence of this shared state, and the serializable guarantees of the CORFU log interface results in a middleware solution that relies on expensive client-to-client communication

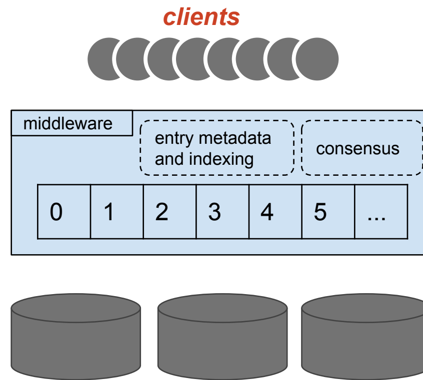


Figure 2.4: Building storage services as middleware requires introducing external services such as consensus, and implementing complex data management tasks like indexing.

for coordination, or external data management services that increase complexity, and may function as a bottleneck on the fast I/O path.

The immutability property of log entries is only one challenge that is difficult with a middleware solution. The CORFU device interface allows any log entry position to be written to any device. This is a property that is used during reconfiguration, when sections of the log address space may be remapped in response to a failure, or an expansion of the set of storage devices. Since the address space of a log in CORFU is the set of 64-bit integers, an efficient index is a necessary requirement to be able to implement the interface. With a middleware-based solution, such an index must be managed as shared state in order for clients to be able to map log addresses onto physical devices. Combined with maintaining per-entry metadata and mechanisms for coordination, entry indexing and remapping services that are needed at a minimum mean that the complexity of middleware solution grows quickly.

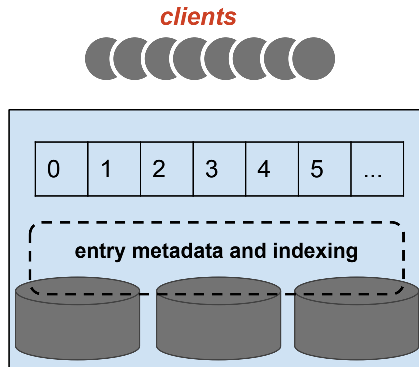


Figure 2.5: By moving key data management responsibilities into the storage device and exposing an application-specific interface, several challenges are solved that existed in a middleware solution.

Device-local coordination

A core challenge with a middleware-based solution is coordinating clients as they access log entries. The method proposed by the CORFU authors is to implement the CORFU I/O interface directly within the storage device. This approach significantly simplifies the construction of the interface because coordination for any single log entry has a natural, well-defined single point in the system.

Figure 2.5 depicts an architecture in which the CORFU interface is implemented within the storage device, containing mechanisms for indexing and coordination. While the CORFU authors have proposed a design for an interface implementation built directly into the device [114], the CORFU authors emulated the interface using a proxy service that exposed an RPC endpoint emulating the interface. In this design, coordination and indexing are managed on top of standard flash devices.

Of course the challenges are not all solved by building storage interfaces directly into devices. Indexing and metadata management must be implemented either in the device, or in a host. Ultimately these interfaces become part of a

larger networked system, with fault-tolerance and reliability challenges that must be solved. Building such a storage interface from scratch requires significant resources and expertise. As we saw with the middleware approach, many services must be combined to build such a system resulting in duplication of services within a larger system context.

2.2.4 Programmability approach

As we saw above, an approach based on middleware poses challenges related to coordination and indexing. And building an interface directly within a storage device is a technique unavailable to all but a select few with the necessary expertise and resources. An approach based on programmability seeks to provide a solution by allowing developers to reuse existing services that solve similar challenges in a way that is approachable to those without significant domain knowledge.

Figure 2.6 depicts a generic storage server that is commonly found in distributed storage systems. While this is a highly simplified view, it serves to highlight common sub-systems and services that exist in such systems. As shown, a server manages one or more storage devices that might include devices for both capacity such as HDDs and latency using flash. A number of internal sub-systems can be found such as indexing, caching, transaction management, I/O and work scheduling, compression, and networking, among a host of others. Each server typically exposes an RPC-like interface for interacting with a block-based or object-based I/O interface, depending on the system.

As discussed above, the CORFU interface generally requires an indexing service, as well as places strict requirements on consistency semantics in order to make guarantees about the behavior of the system. However, as we just described, common server designs in distributed storage systems contain a plethora of sub-

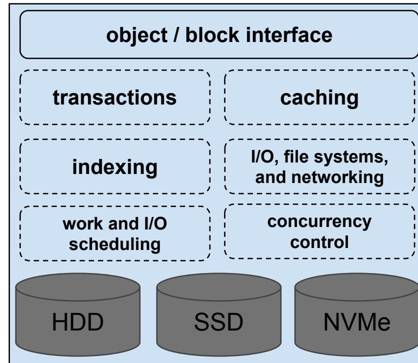


Figure 2.6: Nodes in a distributed storage system typically expose a block-based or object-based interface, and contain numerous sub-systems used to manage data on locally attached storage media.

systems including components such as indexing that are also a key component in building a CORFU storage device interface.

A CORFU storage device interface

We now describe a method by which the CORFU device interface may be instantiated in an existing storage system by reusing existing components. This example is not meant to be exhaustive in its coverage of the CORFU features, but rather to set the context for a discussion about just how large of a design space exists for building such interfaces in existing systems. To this end, we begin with a simple implementation strategy.

The prototype that we describe is built on top of a distributed object storage system, not unlike that which is depicted in Figure 2.6. Object storage systems typically expose a global, flat address space that can be populated with named objects containing user data. Throughout this thesis we use Ceph as a prototyping system, which invariably has its own unique semantics and features.

One simple approach to mapping the log abstraction onto the object storage system in Ceph is to take advantage of the existing interfaces and semantics. For

example, the write-once semantics of log entries in CORFU can be emulated by using the semantics of the existing interface for object creation. In Ceph, object creation can be made exclusive, similar to using `O_CREAT|O_EXCL` when creating a new file in a POSIX file system. By ensuring that all log entry writes are emulated with exclusive object creation, write-once semantics can be achieved by reusing all of the complex machinery in the storage system to make this function behave correctly in a distributed environment. When every log position is mapped to a distinct object, log I/O becomes naturally spread across the cluster, further offloading tasks such as balancing log I/O across the cluster to increase I/O parallelism to the storage system itself.

But constructing an interface like the CORFU *fill* operation is more challenging. Recall that the semantics of the fill interface are that if the entry has been written, the request is rejected. Otherwise, the entry is marked as having been filled, and future reads and writes will be rejected with a status message indicating that it has been filled. The challenge involved in implementing the fill operation arises because the semantics of the interface are defined by domain-specific attributes, and predicates on the current state, and cannot be emulated through mere configuration or composition within the space of existing object interfaces.

To handle this case we advocate that a storage system support the creation of domain-specific object interfaces. In the case of the CORFU interface this would mean the construction of new object methods along side methods like read and write, but designed specifically for the CORFU protocol.

To accomplish this we hook into the existing transaction management subsystem with the Ceph object storage server. This service is used to implement all of the native interfaces, and provides a convenient mechanism to build new interfaces in which existing interfaces can be composed together using standard

C++ and run in an atomic context. The all or nothing atomic semantics provided means that it is easy to keep multiple pieces of data consistent. In the context of CORFU, the fill interface may be implemented by storing a small amount of metadata in the byte stream portion of an object that encodes information about the status of the log entry contained in the object, such as if it is filled, or trimmed. What is interesting is that the entire infrastructure of a full distributed storage system can wrap some isolated C++ code to reproduce some or all of the semantics of an entirely different storage system service.

2.3 Design space challenges

The previous section demonstrated some of the benefits of using programmability to implement application-specific I/O interfaces. In particular, it was shown how services found within Ceph could be reused to build the storage device interface used by the CORFU protocol. However, only a single implementation was described, and in general there may be many ways to build an implementation providing the same semantics and optimization strategies. In this section it will be shown that the size of the design space that contains such interface implementations is large, and difficult to navigate without automated assistance. These example implementations are designed to represent approaches that may likely be considered by someone with a reasonable familiarity with Ceph as a platform for building new I/O interfaces.

2.3.1 Performance portability

Let us take a step back now from the high-level sketch of an implementation of the CORFU interface described in the previous section. In reality there are

many ways to construct interfaces in a system. In order to make this concrete, consider the Ceph object storage system that we are using as a prototyping vehicle throughout this thesis. At the bottom of Figure 2.7 we have enumerated just a few of the different internal storage sub-systems found within the Ceph storage system.

For example, Ceph may be configured to store objects in a file system, in an embedded database such as RocksDB or LMDB, or a new sub-system called Bluestore that is designed to take advantage of HDDs as well as flash media and NVMe. In addition, there are multiple data management systems contained internally. Bulk data is often stored using an interface such as a file byte stream that is optimized for large contiguous chunks of unstructured data. But the system also allows an arbitrary number of key-value pairs to be associated with each object, providing an embedded database-like interface in addition to the byte stream interface. These remain the same even when the underlying implementation of the interfaces change, or different hardware is deployed. These interfaces are discussed further in the next chapter.

In the previous section the CORFU interface was implemented on the storage system by storing each log entry in a single, uniquely named object. This type of partitioning is often referred to as 1-1, and may not be an optimal strategy for some systems. For example managing a huge number of small objects is not the most efficient way to store data in Ceph. At the top of Figure 2.7 we highlight a second form of partitioning labeled *striping*, in which the log is distributed in a round-robin fashion across a fixed number of objects. This is referred to as an N-1 strategy in which multiple log entries are stored in a single object.

Using just two dimensions, low-level I/O sub-system (key-value interface or bytestream interface for entry storage), and high-level log partitioning strategy,

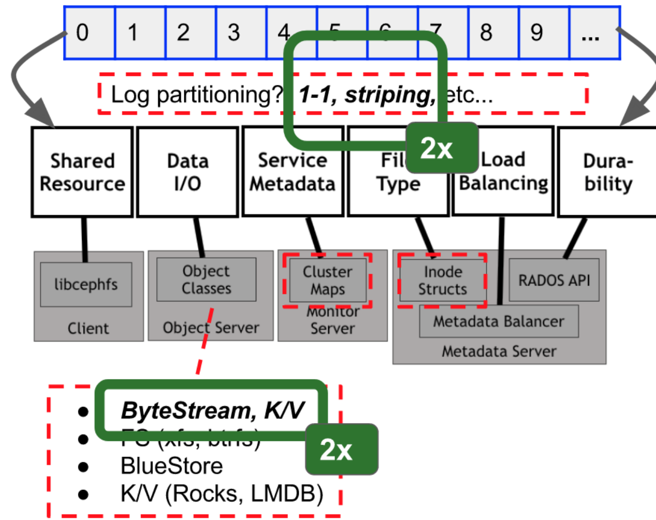


Figure 2.7: Four different strategies for implementing the CORFU interface are considered along two dimensions: low-level I/O interface and high-level log partitioning strategy.

we can identify four different candidate implementation strategies of the CORFU interface in Ceph. We implemented each of these strategies, and we found that even routine software upgrades can cause performance regressions which manifest as obstacles for adopters of a programmable approach to storage interface development.

Figure 2.8 shows the append throughput of the four implementations running on two versions of Ceph, one from 2014 and one from 2016. Each of the four lines represents a different implementation. The experiments are all using the same hardware, in which the performance in general can be seen to be significantly better in the newer version of Ceph from 2016. However, if we consider other costs such as software maintenance then these results reveal a trade-off.

First, notice that in 2014 (bottom) the top two performing implementations achieve roughly the same throughput performance, within a few percentage points. However, they differ in their implementation complexity. For example, more work is involved in managing round-robin striping across a fixed number of objects.

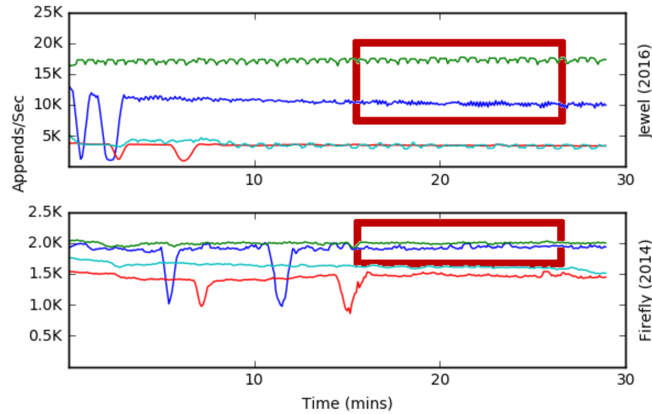


Figure 2.8: Append performance of four shared-log implementations on two versions of Ceph before and after a major release upgrade using the same hardware.

We make the observation that an application developer in 2014 may have made essentially a random decision between these two implementations. Each exhibit fairly poor performance, and choosing the simpler implementation for little gain would be reasonable.

When we repeat the experiments using the same hardware and the same implementation, but with a newer version of Ceph a challenge presents itself. Notice that the difference in performance between these two implementations in 2016 are, in comparison, significantly different. A developer in 2014 that chose the blue implementation will be losing out on a lot of throughput after a routine upgrade. But what is more insidious is that given the large performance increase after the upgrade, a developer may not even know that a separate implementation will provide even greater throughput.

This example is about as simple as one can construct, as it ignores tuning parameters and a vast sea of configuration options. Choosing the best implementation of a storage interface depends on the timing of development (e.g. system version); the expertise of programmers and administrators; tuning parameters and hardware configuration; physical design and the use of internal I/O inter-

faces; and system-level and application-specific workload characteristics. A direct consequence of such a large design space is that some choices may quickly become sub-optimal as aspects of the system change. This forces developers to revise implementations frequently, increasing the risk of introducing bugs that, in the best case, affect a single application and, in monolithic designs, may cause systemic data loss.

2.3.2 Complexity

The CORFU use case that we have been examining is a fairly narrow example in that it depends on a small portion of an otherwise large system. Building an interface that uses many more internal services, and exploring each implementation option under different hardware and software configurations would be a massive undertaking. It is relatively easy to highlight how much additional complexity exists within these systems, and quickly arrive at the observation that this complexity largely prevents human developers from being able to handle all of the optimization scenarios, even for static design cases. And given the differences between storage systems, it is unlikely that anything more than a general implementation strategy would be portable; in a new system the entire design process may need to be repeated.

Hardware

Ceph is a software-defined storage system, which means that its features are primarily derived from software implementations, and as such will run on a large variety of hardware. Each hardware configuration encompasses specific sets of performance characteristics and tunables (e.g. I/O scheduler selection, and policies such as timeouts). For example, Ceph can currently run on a mixture of slow

spinning disk, SSDs, and NVMe simultaneously, using each device for a different type of data amenable to the performance and capacity of the device itself.

Ceph has been deployed on an interesting variety of hardware as well. A recent example is a deployment of 500 network attached Ethernet hard drives that contain enough local compute and memory resources to host a full Ceph storage node [26]. At the other end of the spectrum Ceph has been deployed as a single 65 petabyte cluster across 10,800 storage nodes [27]. But its deployment is more common on standard commodity hardware in clusters that are smaller than 10,000 nodes.

In our experiments, we tested a variety of hardware and discovered a wide range of behaviors and performance profiles. While we generally observe expected improvements on faster devices, choosing the best implementation strategy is dependent on hardware and the strategy by which data is stored. This will continue to be true as storage systems evolve to support new technologies such as persistent memories and RDMA networks that may require entirely new storage interfaces for applications to fully exploit the performance of next-generation hardware.

Software and tunables

The primary source of complexity in large storage systems is, unsurprisingly, the vast amount of software written to handle challenges like fault-tolerance and consistency in distributed and heterogeneous environments. In our experiments we encountered performance portability challenges simply as a result of a routine software upgrade.

To put the complexity in context, a recent version of Ceph from May 2018 had approximately 1400 tunable parameters controlling all aspects of the system. This includes for example 163 parameters controlling different aspects of the metadata

service including the file system client. Over 160 tunables control various aspects of the Ceph monitoring, Paxos, authentication, and encryption sub-systems. And over 500 tunables were found related to the core object storage server daemons and its various storage backends including sub-systems such as RocksDB and LevelDB. These parameters are far reaching in scope, controlling sensitive aspects of performance such as timeouts, threading, and message queueing. This is only a subset of the parameters, and doesn't include aspects of networking for which there are multiple implementations including an asynchronous messaging system and a messaging system that supports RDMA acceleration.

Evolving hardware, software, and system tunables presents a challenge in optimizing systems, even in static cases with fixed workloads as we have seen. Programmable storage approaches that introduce application-specific interfaces are sensitive to changes in workloads and the cost models of low-level interfaces which are subject to change. This greatly increases the design space and set of concerns that must be addressed by programmers.

Conclusion

The availability of high-performance open-source storage has prompted developers to explore new, domain-specific storage interfaces that replace or augment standardized interfaces like POSIX. As this trend continues to grow, more and more software will be written with dependencies on custom interfaces that may have implementations in only one storage system. When these implementations depend on assumptions about system-specific internal features, portability becomes an important challenge. And as we have seen, the performance profile of internal interfaces can change from routine upgrades.

Despite the benefits, the overhead of maintenance can be a limiting factor. We

therefore propose that interfaces be built in a way that reduces the overhead of maintaining interface implementations using techniques drawn from the literature on declarative languages and database optimization. In the next section we will explore one potential direction for this research.

2.4 Declarative storage

We believe a better understanding of application and interface semantics exposes a frontier of new and better approaches with fewer maintenance requirements than hard-coded and hand-tuned implementations. An ideal solution to these challenges is an automated system search of *implementations*—not simply tuning parameters—based on programmer-produced specifications of storage interfaces in a process independent of optimization strategies, and guaranteed to not introduce correctness bugs.

Despite the benefits of the approach to building new interfaces, the technique requires navigation of a complex design space while simultaneously addressing often orthogonal concerns (e.g. functional correctness, performance, and fault-tolerance). Worse still, the availability of domain expertise required to build a performant interface is not a fixed or reliable resource. As a result, interfaces become sensitive to evolving workloads. This results in burdensome maintenance overhead when underlying hardware and software changes.

To address these challenges, we advocate for the use of high-level declarative languages (e.g. Datalog) as a means of programming new storage system interfaces. By specifying the functional behavior of a storage interface once in a relational (or algebraic) language, optimizers built around cost models can explore a space of functionally equivalent physical implementations. Much like query planning and optimization in database systems, this approach will logically dif-

ferentiate correctness from performance, and protect higher-level services from lower-level system changes [95]. However, despite the parallels with database systems, fundamental differences exist in the optimization design space.

Although powerful, storage interface construction in the way we advocate is a double-edged sword. The narrowly-defined interfaces dominating systems today have been a boon to developers by limiting the size of the design space where applications couple with storage, allowing systems to evolve independently. Programmable storage lifts the veil on the system and, thereby forces developers of higher-level services to confront a much broader set of possible designs.

2.4.1 The Bloom language

Current ad hoc approaches to programmable storage restrict use to developers with distributed programming expertise, knowledge of the intricacies of the underlying storage system and its performance model, and use hard-coded imperative methods. This limits the use of optimizations that can be performed automatically or derived from static analysis. Based on the challenges we have demonstrated stemming from the dynamic nature and large design space of programmable storage, we propose an alternative, declarative programming model which reduces the learning curve for new users, and allows existing developers to increase productivity by writing fewer, more portable lines of code.

The model we propose corresponds to a subset of Bloom, a declarative language for expressing distributed programs as an unordered set of rules [13]. Bloom rules fully specify program semantics and allow developers to ignore the details associated with program evaluation. This level of abstraction is attractive for building storage interfaces whose portability and correctness is critical. We use Bloom to model the storage system state uniformly as a collection of relations,

with interfaces expressed as a collection of *queries* over a request stream that are filtered, transformed, and combined with other system state.

2.4.2 Declarative CORFU protocol

We can model the storage interface of the CORFU protocol as a query in Bloom in which the shared-log and metadata are represented by two persistent abstract collections mapped onto physical storage. Listing 2.1 shows the specification of state for the CORFU interface. Lines 2 and 3 define the schema of the two persistent collections that hold the current epoch value, and the log contents. These collections are mapped onto storage within Ceph but abstract away the low-level interface (e.g. bytestream vs key-value). That is, neither table definition imposes restrictions on the internal storage interfaces used, a decision that we saw can cause challenges when upgrading software which changes the performance profile of internal interfaces. Lines 5-9 define the input and output interfaces. We use a generic schema for the input operation to simplify how rules are defined that apply to all operation types if the CORFU interface. Lines 11-15 define named collections for each operation type. The scratch type indicates that the data is not persistent, and only remains in the collection for a single execution time step. This property can be useful to an optimizer in selecting where data is stored and how it should be treated. The remaining scratch collections are defined to further subdivide the operations based on different properties which we'll describe next.

Initialization is performed in Listing 2.2 which acts as a demux for the operation type and properties. Lines 3-7 show the epoch guard that is applied to all operations. The guard rejects requests that are tagged with old epoch values, ensuring that a client generating a request has an up-to-date view of the system. First the `invalid_op` collection is defined to include the current operation if its

```
1 state do
2   table :epoch, [:epoch]
3   table :log, [:pos] => [:state, :data]
4
5   interface input, :op,
6     [:type, :pos, :epoch] => [:data]
7
8   interface output, :ret,
9     [:type, :pos, :epoch] => [:retval]
10
11   scratch :write_op, op.schema
12   scratch :read_op,  op.schema
13   scratch :trim_op,  op.schema
14   scratch :fill_op,  op.schema
15   scratch :seal_op,  op.schema
16
17   # op did or did not pass the epoch guard
18   scratch :valid_op,   op.schema
19   scratch :invalid_op, op.schema
20
21   # op's position was or was not found in the log
22   scratch :found_op,    op.schema
23   scratch :notfound_op, op.schema
24 end
```

Listing 2.1: Data structures used in the Bloom specification of CORFU.

epoch value is no larger than the stored epoch value in the epoch table. Next the `valid_op` collection is defined to be the inverse of `invalid_op` and is a helper used to refine other operations later in the dataflow. Finally we handle the case for all operations tagged with an out-of-date epoch by merging the `invalid_op` set into the output `ret` collection.

Lines 10 and 11 populate the `found_op` and `notfound_op` collections that allow operation behavior to be predicated on if the position associated with a request is found in the log. Finally the remaining lines in Listing 2.2 populate each of the specific operation collections.

The process of sealing an object requires installing a new epoch value and returning the current maximum position written. Listing 2.3 implements the seal interface by first removing the current epoch value and replacing it with the epoch value contained in the input operation. Next an aggregate is computed over the log to find the maximum position written, and this value is returned, typically to

```
bloom do
  # epoch guard
  invalid_op <= (op * epoch).pairs{|o,e|
    o.epoch <= e.epoch}
  valid_op <= op.notin(invalid_op)
  ret <= invalid_op{|o|
    [o.type, o.pos, o.epoch, 'stale']}

  # op's position found in log
  found_op <= (valid_op * log).lefts(pos => pos)
  notfound_op <= valid_op.notin(found_op)

  # demux on operation type
  write_op <= valid_op {|o| o if o.type == 'write'}
  read_op  <= valid_op {|o| o if o.type == 'read'}
  fill_op  <= valid_op {|o| o if o.type == 'fill'}
  trim_op  <= valid_op {|o| o if o.type == 'trim'}
  seal_op  <= valid_op {|o| o if o.type == 'seal'}
end
```

Listing 2.2: Initialization steps in the CORFU Bloom specification.

```
bloom :seal do
  epoch <- (seal_op * epoch).rights
  epoch <+ seal_op { |o| [o.epoch] }
  temp :maxpos <= log.group([], max(pos))
  ret <= (seal_op * maxpos).pairs do |o, m|
    [o.type, nil, o.epoch, m.content]
  end
end
```

Listing 2.3: CORFU Seal implementation in Bloom.

a client performing a reconfiguration of the system or following the failure of a sequencer.

Trimming a log entry always succeeds. In Listing 2.4 the `<+-` operator simultaneously removes the log entry with the given position and replaces it with an entry with its state set to *trimmed*. In practice the removal of a log entry may trigger garbage collection, but we model it here as an update for brevity.

The write and fill interfaces are implemented similarly, and are both shown in Listing 2.5. A `valid_write` collection is created if the operation position is not found in the log. A `valid_write` operation is then merged into log, otherwise a *read only* error is returned indicating that the log position was already written to. The fill operation is identical except the fill state is set on the log entry.

```
1 bloom :trim do
2   log <+- trim_op{|o| [o.pos, 'trimmed']}
3   ret <=  trim_op{|o|
4     [o.type, o.pos, o.epoch, 'ok']}
5 end
```

Listing 2.4: CORFU Trim implementation in Bloom.

```
1  bloom :write do
2    temp :valid_write <= write_op.notin(found_op)
3    log <+ valid_write{ |o| [o.pos, 'valid', o.data]}
4    ret <= valid_write{ |o|
5      [o.type, o.pos, o.epoch, 'ok'] }
6    ret <= write_op.notin(valid_write) {|o|
7      [o.type, o.pos, o.epoch, 'read-only'] }
8  end
9
10 bloom :fill do
11   temp :valid_fill <= fill_op.notin(found_op)
12   log <+ valid_fill { |o| [o.pos, 'fill'] }
13   ret <= valid_fill { |o|
14     [o.type, o.pos, o.epoch, 'ok'] }
15   ret <= fill_op.notin(valid_fill) { |o|
16     [o.type, o.pos, o.epoch, 'read-only'] }
17 end
```

Listing 2.5: CORFU Write and Fill implementations in Bloom.

Finally the read interface is shown in Listing 2.6, and structured in a similar way to the write and fill interfaces. First we create a collection containing a valid read operation (named `ok_read`) that is in the log and does not have the filled or trimmed state set. The data read from the log is returned in the case of a valid read operation, otherwise an error is returned through the output interface.

This transformation from hard-coded interfaces into declarative specification permits optimizations and implementation details (e.g. log striping and partitioning) to be discovered and applied transparently by an optimizer. Since the specification of the interface is invariant across system changes and low-level interfaces, an optimizer can automatically render execution decisions and build indexes using the performance characteristics of specific access methods.

Amazingly, the semantics of the entire storage interface requirements in CORFU are expressible using only a few Bloom code snippets, not any more complicated

```
bloom :read do
  temp :ok_read <= (read_op * log).pairs(pos => pos) { |o, l|
    [o.type, o.pos, o.epoch, l.data] unless
      ['filled', 'trimmed'].include?(l.state) }
  ret <= ok_read { |e|
    [e.type, e.pos, e.epoch, e.data] }
  ret <= read_op.notin(ok_read, type=>type) do |o|
    [o.type, o.pos, o.epoch, 'invalid']
  end
end
```

Listing 2.6: CORFU Read implementation in Bloom.

than the example above, which are amenable as input to an optimizer. Beyond the convenience of writing less code, the entire experience of designing and writing an interface such as CORFU in a declarative language such as Bloom eases the process of constructing convincingly correct implementations. Specifically, the high-level details of the implementation mask distracting issues related to the physical design and the many other "gotchas" associated with writing low-level systems software.

2.4.3 Sequencer specification

Our current Bloom specification of CORFU assumes the existence of an external sequencer service to assign log positions. A declarative specification of the sequencer will be critical to providing portability of the service, since performance relies on optimizations enabled by volatile memory, fast fail-over, and configuration management of the sequencer depends on an auxiliary service such as Paxos. Since distributed storage systems internally utilize volatile storage in many forms (e.g. memory caches and non-replicated data), and tend to use systems like Paxos for state management, we seek to avoid replicating these features. We will address this in Sections 4.7 and 5.5.1 in which we examine how components such as the file system metadata service, as well as the object storage interfaces of the storage system can be reused to construct a high-performance sequencer.

2.4.4 Offline optimization

Our discussion about declarative languages and the use of database optimization techniques raise many valid concerns. For example, the use of optimization techniques for query planning in database management systems are typically assumed to be online optimizations that occur for every query executed. Expecting a distributed storage system to perform query optimization for every I/O operation raises important performance and feasibility concerns. As discussed above, data layout and the interfaces used to store data can be have a significant impact on performance. Frequent changes to data layout, or moving data between interfaces can represent major challenges to performance management.

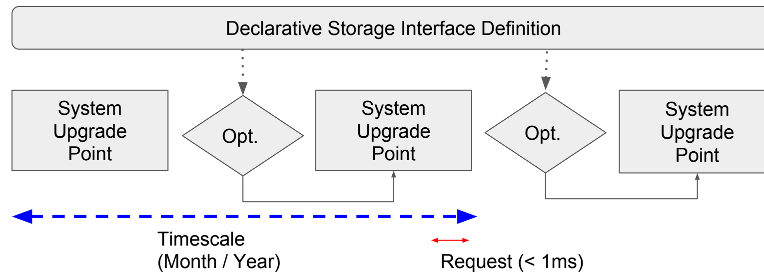


Figure 2.9: New interfaces are installed at longer timescale than that of individual I/O operations in a system and can correspond to well-defined system life cycle events such as hardware or software upgrades.

We address these concerns by observing that the primary performance changes we observed occurred on well-defined boundaries related to hardware and software upgrades. These types of changes occur on a timescale that is orders of magnitude larger than that of individual I/O operations. Therefore our work takes advantage of the flexibility afforded by offline optimization strategies. This is graphically shown in Figure 2.9 in which optimization and query planning for declarative interface specifications are handled between upgrade points as part of the standard upgrade process, which can be, and often is, planned for and scheduled ahead of

time. By producing static implementations of interfaces periodically we retain the benefit for developers of using a declarative language that avoids the pitfalls of coding to low-level interfaces, while avoiding the challenges associated with online optimization.

2.5 Conclusion and scope

Storage system programmability is a trend that is emerging quickly, and has the potential to transform the way data-intensive application are built. By leveraging domain-specific interfaces that are tailored to an application, system design can be a direct reflection of the application semantics rather than a collection of abstraction layers and external services combined in ad hoc ways. There are many real-world examples of programmability in use, and we have explored one powerful use case with the CORFU log abstraction. However, by avoiding the use of standardized interfaces we can observe that challenges in software maintenance and performance portability will be a major challenge for gaining adoption. This is due the extremely large design space that exists for developers of new storage interfaces using programmability techniques.

The discovery of such a large design space was made at a late stage in development of this thesis. However, as discussed in this section, navigating the design space is a challenge that must be addressed for the success of programmable storage as a paradigm for building new I/O services. We believe, and there is strong evidence for, the use of declarative languages in expressing I/O services in a way that allows the existing body of work in compiler and database optimization techniques to be applied.

That said, the scope of this thesis is restricted to demonstrating that a variety of I/O services can be expressed declaratively in such a way as to support known

optimization strategies. Throughout the thesis we will show how a declarative specification may be arrived at for the various use cases that explore, and how an optimizer may reasonably be able to make important optimization decisions. This thesis does not address many of the challenges related to designing and implementing a full language, compiler, and run-time system that will be needed in a real world system that is integrated into a production distributed storage system. That is left to future work.

Chapter 3

Related Work

3.1 Active storage

There has been a wide range of research related to the topic of co-locating computation with data, generally referred to as active storage. In each instance the set of target optimizations include exploiting excess parallelism, and reducing data movement by exploiting computational resources located near target data. Riedel, et al. [91] introduced the active storage concept by offloading application functions to on-disk microcontrollers, taking advantage of spare cycles as well as reducing data sent over the host bus. Addressing concerns over the safety of offloading arbitrary functions, the Active Disk project introduced a stream-based programming model, and further restricted the execution environment by preventing memory allocation and direct initiation of I/O. Extensions to the active storage concept have used disk-to-disk communication to avoid bus communication [70], as well integrating additional resources that fall along the I/O path such as disk array controllers [103]. Recently the active storage concept has been applied to solid-state devices using customizable firmware and focuses on energy reduction [110]. Most recently active storage has been revisited in the context of

solid-state drives in which resources within the SSD are exploited to perform work on behalf of client applications. This has been examined in the general context of how to build such SSD applications [96], as well in the context of database query acceleration [66, 42].

The use cases of programmability when constructing data interfaces are motivated in a similar way to many techniques like active storage which seek to exploit remote resources and reduce data movement. The work we have presented make use of host resources, but could also integrate resources found on disks and media themselves. We also note that these modes of exploiting programmability are not our focus, but rather driving examples.

3.1.1 Object-based storage

Du [45] proposed Intelligent Storage for object-based storage devices (OSDs) which could take advantage of not only low-level resources like on-disk controllers, but also more powerful host CPUs and larger memory, effectively utilizing more of the compute center hierarchy. This concept has been applied by extending the iSCSI standard to attach functions written in Java to object attributes and allow these functions to be executed remotely [67]. Xie, et al. [123] proposes extensions to the T10 standard for associating vendor-authored active storage functions with special "function objects", and the ability to compose functions remotely. While Xie, et al. address security concerns by restricting functions to be vendor provided, Runde, et al. provides security using sandboxing virtualisation technology to allow arbitrary code to be executed remotely [93].

For example, the state-of-the-art V8 JavaScript runtime has been embedded into RAMCloud [87] to to provide optimizations such as remote use of CPU resources, as well as optimizations such as remote pointer chasing [125]. The Comet

system is a distributed key-value storage system designed for extensibility. It includes the concept of active storage objects which can be programmed to react to changes in the system [49]. The Comet system is able to customize many aspects of the system such as replication, and implement functions in which object handlers communicate with each other. The Swift distributed object storage system supports the concept of a Storlet which is a function that can be attached to an object and triggered by various life cycle events [90]. Storlets use Docker for isolation.

The programmability features we have built into Ceph depend on a form of sandboxing using the Lua programming language. Our consideration of Lua is functional and meant to ease the process of exploring the design space, rather than based on a need for absolute security. Existing work on sandboxing can be applied in our case using other languages when needed. Related conceptually to our discussion of management of code and interfaces in Chapter 6, Xie, et al. [123] take steps with their "function objects" for managing code implementing object interfaces. They use the objects directly, while we propose using cluster-level metadata services.

Recently storage device manufacturers have been exploring the use of embedding more powerful embedded processors, DRAM, and flash to construct so-called *smart drives*, such as the Kinetic drives from Seagate [4]. These drives present a key-value interface, but also contain higher-level primitives such as moving data between drives across the network which allow more complex data management scenarios to be constructed.

3.1.2 Domain-specific interfaces

Active storage concepts have been applied in several domain-specific contexts. Uysal, et al. compares the use of active storage for accelerating decision support databases against other scalable architectures [112]. In [35] Chiu, et al. simulated an active storage environment with offloaded database operations (e.g. scan, join) and a 2-D fast Fourier transform using algorithms specifically tailored for the distributed, active storage environment. Huston, et al. [60] exploited the properties of search tasks to optimize a programming model for early discard filtering, observing that locking could be simplified due to the read-only nature of search, objects could be processed in any order, and avoiding persistent state in the protocol. Lim, et al. [76] applied active storage to build the Active Disk File System which offloaded core file system functionality (e.g. file lookup operations) to low-level active components. Recently, Gkantsidis, et al. [51] used static analysis to automatically extract early discard filters from Hadoop MapReduce jobs and apply the filtering transparently within cloud storage devices.

Caribou is a distributed storage system that exposes a key-value storage interface and integrates FPGAs and low-level optimized data structured for accelerating database query processing [65]. The system uses specific access methods designed for database queries. Wei, et. al [115] demonstrate how to build a shared-log interface directly into SSDs in support of high-level domain-specific interfaces the shared-log CORFU abstraction. Unlike our work, these systems do not examine how to use existing system components to construct new services in a distributed system.

More recently, the Department of Energy sponsored FastForward I/O Initiative has proposed next-generation I/O interfaces that include native support for non-POSIX byte stream interfaces, as well as multi-dimensional array objects that

support active storage operations such as data analysis and subsetting.

Both Boxwood and Sinfonia [82, 12] are storage systems that expose building blocks to application developers. Sinfonia exposes raw memory address spaces, and allows applications to create new functionality by supporting efficient transactions over storage. Boxwood is a storage system that exposes a set of abstractions such as distributed trees and hash-tables that can be recombined by applications to create new functionality while sharing the underlying sub-system implementations. Both systems expose services at a much lower-level of abstraction than we have explored in this thesis, and both are new storage systems rather than exploring the reuse of existing subsystems found in an existing system.

3.1.3 Parallel file systems

Applying active storage concepts to parallel file systems is more challenging because data may be distributed across many distinct storage resources. In [46], Felix, et al. introduced an in-kernel interposition layer in Lustre that redirected operations to a user-space execution engine that provided support for stream-based data transformations. Piernas, et al. extended the work in [46] to provide an entirely user-space solution that offered additional flexibility over the kernel-based version. Striped data was handled transparently by the execution environment on a node by remotely reading required non-local data, creating an I/O bottleneck. Piernas, et al. [88] addressed striped, complex file formats such as netCDF by using knowledge of striping strategy and file layout information contained in the netCDF header to only perform processing on nodes known to contain target data. Son, et al. [104] extended the MPI-IO programming model to include collective operations (e.g. global reduction) over the MPI-IO data model using server-side collectives implemented in PVFS.

3.1.4 Performance management

While we did not address performance management in this work, it is a component of future work and an important topic to cover when considering how to construct a programmable storage system. The following work primarily focus on the placement of functions that use resources which is an important component of any cost model associated with function shipping.

Limited work has addressed the challenge of performance management in the context of active storage, and has focused on the problem of function placement. That is, optimizing the location that a function is executed at (e.g. remote host vs. local client) for a particular policy. All approaches are similar in that they rely on a unit of functionality (e.g. a primitive operation or computational kernel) being flexibly scheduled. Amiri, et al. [15] constructs a graph of application function dependencies, and utilizes collected run-time statistics to model alternative function placements. In [122] Wickremesinghe, et al. construct a programming model specifically designed for I/O efficient algorithms that balance primitive operations specific to the problem space between clients and hosts. In the context of database systems, Qiao, et al. offload database functionality into an active storage system, and integrate the cost of remote execution into the database query optimizer to intelligently schedule function placement [89]. In [34], Chen, et al. introduce the DOSAS system that treats function placement and request ordering within storage nodes as a binary optimization problem and uses detailed statistics to construct schedules that minimize overall execution time. Chen, et al. [33] consider the problem of data dependence when scheduling function placement by predicting the alignment of functions with low-level storage partitions, and by extension the associated cost of performing remote I/O operations.

3.2 Storage system extensibility

Several file systems have been proposed that offer alternatives to the standard interpretation of a file as a single, linear byte-stream. Kots, et al. propose a file system interface for multi-processors that define access to both bytes and records, and allow a file to represent a collection of subfiles useful for parallel applications [74]. Corbett, et al. proposed the Vesta File System for exposing available parallelism within the storage system to application [37]. A file in Vesta is defined by two-dimensions, one controlling parallelism by addressing physical storage partitions, and a second dimension controlling the layout of fixed-size data elements in each partition. The Galley File System introduced by Nieuwejaar, et al. is similar in Vesta, but adds a third dimension that allows for multiple, variable-size subfiles within each physical partition, supporting a wider range of striping patterns that could be expressed [85]. Finally, Isaila, et al. develop a language called *PITFALLS* used in the Clusterfile file system for expressing a wide range of patterns that define physical partitions and logical views of files composed of fixed-size records [63].

Several proposals have been put forth to create file systems whose functionality can be extended. The Extensible File System (ELFS) proposed by Karpovich, et al. is a methodology for constructing interfaces based on the object-oriented paradigm [69]. Interfaces in ELFS are middleware objects (e.g. 2DArrayFile) that implement a file format on top of an underlying file system interface. Haines, et al. [56] introduce the SmartFiles concept that use type definitions expressed in the DAFT (Data File Types) language, combined with run-time parameters such as the current process identifier to implement file with domain-specific interfaces. Today, SmartFiles closely resemble popular serialization technologies such as Avro [1], Protocol Buffers [8], and MessagePack [5].

To address the challenge of exploring changes to core file system functionality and semantics, Huber, et al. introduced the PPFS file system which is a middleware layer designed to ease the exploration of the I/O system design space [59]. The middleware layer wraps a traditional file system, and allows customization of different policies such as striping, data partitioning, and caching. Taking extensibility to its near logical extreme, the Hurricane File System defines a file in terms of a composition of fine-grained building blocks that control functionality such as data layout, replication, locking, caching, prefetching, and authentication. And more recently, Grawinkel [53] embedded a Lua interpretor inside pNFS clients to allow scriptability of file layouts in terms of target storage targets. GlusterFS [3] translators provide a rich mechanism for adding functionality at different levels of the file system, the are intended to be used to create long-lived extensions.

Much of the existing work related to extensibility is complementary to what we present in this thesis. The lessons learned from this work, including especially the modes by which new extensions are expressed may help define our future in declaratively specifying new storage interfaces. Finally, systems like Hurricane which are similarly motivated as our work take the approach of building an entirely new system to demonstrate extensibility rather than examining the state of existing storage systems as a set of building blocks.

3.3 Software-defined storage

Software-defined storage (SDS) is a term to describe the use of abstractions found in storage systems that separate the data plane from the control plane and allow aspects of storage systems to be controlled using configuration settings and policies. Early work on SDS includes IOFlow [109] that allowed I/O requests flowing through the system to be identified using a tuple specification consisting of

I/O request properties, and map these requests to properties such as I/O priority. The sRoute project [106] extended the work of IOFlow allowing I/O requests to be flexibly routed within the storage system to control behavior for tenants such as routing requests to achieve tenant cache isolation. Generally software-defined storage systems today are used to provide functionality such as resource provisioning, in contrast to the use of programmability for creating entirely new functionality.

The CoSS storage system is an HPC storage system that allows applications to customize its behavior using contracts [44]. A contract declaratively describes the data model used by an application, as well as views that describe what data will be stored and what data will later be read. Using information from the contract the storage system can decide when and where to perform data transformations and what form data should be stored in. In certain aspects this work is related to our future goals of declarative storage, and is likely complementary although as proposed the specifications of CoSS are at a much higher-level of abstraction providing coarser-grained specializations.

The Crystal storage system is closely related to the work present in this thesis [52]. The goal of Crystal is similar which is to reduce or eliminate hard-coded functionality in the system, and enable to new functionality programmatically, handling the needs of applications that change over time. Crystal introduces abstractions such as filtering, controllers, and policies and uses a domain-specific language for expressing composition of these components. This work is complementary to ours. Crystal allows new application-specific operations to be created on objects, but Crystal is a storage system built from the ground-up using a variety of separate systems, rather than examining how their work could be integrated into an existing storage system by using reusable components.

The Mochi project is an effort related to the Exacale initiative in the U.S. to create a platform for building application-specific storage interfaces for HPC use cases by combining system building blocks [43]. The building blocks that Mochi uses are low-level, including things like RDMA networking, RPC, threading and memory models, and embedded databases and virtual machine interpreters. Like Crystal, Mochi creates a new set of services built from the ground-up or repurposed from other systems rather than exploring the re-use of sub-systems found within an existing system. Mochi is highly relevant to our future work with declarative storage systems. For example, Mochi has been exploring the creation of interfaces for reusable systems that support composition.

Chapter 4

Data interfaces

The data interfaces exposed by a storage system are the primary interaction points with applications and services. Examples of data interfaces include POSIX files, virtual block devices, and object-based storage. Fundamentally a data interface is a means by which the storage system exposes internal services and resources. For example, storage systems often provide durability guarantees to applications writing to POSIX files by transparently replicating data providing redundancy in case of failure. Storage systems may also reveal details about the underlying physical system through data interfaces; it is common for object-based storage systems to map objects onto a single storage device, allowing applications to reason about I/O parallelism. Storage systems today tend to expose a fixed set of interfaces that target broad categories of applications, and as such are often very general in nature. This is useful for storage system developers as the scope of possible data interfaces is bounded, as well as for developers building high-level applications and services that can benefit from the portability of data interfaces. However, applications and data management services have a very rich and diverse set of storage requirements and domain-specific behaviors. When a limited set of storage interfaces is available, this results in additional costs associated with

storage-related development tasks to map between semantics and work around limitations of existing interfaces.

Programmable storage systems address this by allowing new data interfaces to be constructed by programmatically combining internal services and resources. This is useful for building interfaces that provide application-specific semantics, can simplify application design, and provide unique optimization opportunities.

4.1 Overview

In this chapter we are going to provide a sample of the breadth of applications that can benefit from programmable data interfaces. However, as we will show, the current state of building data interfaces is difficult, time consuming, and the results challenging to maintain and migrate to new systems. Throughout this section we will highlight these challenges and argue that a declarative approach to building data interfaces is a viable option for building future programmable storage systems.

The methodology we use in this chapter is to select a set of applications that demonstrate broad categories of application needs. For each use case covered in we show how programmability can be used beneficially, summarize major challenges, and show how declarative approaches can help resolve these issues. This process is a systematic exploration of the design space of programmability in distributed storage systems. We examine interfaces for transactional data management, query acceleration in database management systems, and structured data management common in scientific computing. In addition, we show how durability properties of a storage interface can be used to create domain-specific optimizations, and how taking advantage of data availability mechanisms, fault-tolerant network services can be created using data interface programmability.

First we provide an overview of programmability in Ceph which is our primary prototyping platform, and then discuss the general challenge of physical design when building new data interfaces. A primary goal of this chapter is to chart a course for future work in programmable storage by showing the breadth of applications that can benefit. Each subsequent section will cover one application and use case in detail. Each section contains a brief overview and discussion that relates the findings to declarative storage techniques as discussed in Section 2.4.

4.2 Programmability

4.2.1 Ceph

We utilize the Ceph storage system as a prototyping platform for developing new data interfaces. Ceph already provides a form of programmability with a feature called object interface plugins. An object interface is a plugin structured similarly to that of an RPC in which a developer creates a named function that exists in and is managed by the storage cluster which clients may remotely invoke. In the case of Ceph each remote function defined by a plugin is implicitly run within the context of a single object specified when the function is called. In effect, these plugins allow developers to add new interfaces to object-based storage. For example, using these feature an application developer could create a new function that remotely computed the MD5 hash of an object and invoke that method remotely without returning all of the data and computing the hash locally. Developers of object interfaces express behavior by creating a composition of existing native interfaces, or other custom object interfaces, and handle serialization of function input and output. A wide range of native interfaces are available to developers such as reading and writing to a byte stream, controlling object snap-

shots and clones, and accessing a sorted key-value database associated with each object, among others. One of the most powerful features that Ceph exposes is that the composition of these native interfaces may be transactionally composed along with application specific logic to create semantically rich interfaces. An example of this would be an interface that atomically updates a matrix stored in the byte stream and an index of the matrix stored in the key-value database. Without the ability to atomically update independent pieces of data, applications may be forced to use heavyweight, and difficult solutions like two-phase commit to maintain consistency.

The implementation of Ceph's object abstraction, although powerful, does not support programmability in a convenient form. Supporting only C/C++ for object interface developers, Ceph requires distribution of compiled binaries for the correct architecture, adding a large barrier of entry for developers and system administrators. Second, having no way to dynamically unload modules, any changes require a full restart of a storage daemon which may have serious performance impacts due to the loss of cached data in volatile memory. And finally, the security limitations of the framework limit the use of object interfaces to all but those with administrative level access and deep technical expertise.

To address these concerns, we have developed Lua [80] extensions that allow new object interfaces to be dynamically defined, loaded into the system, and modified at runtime, resulting in an object storage API with economy of expression, which at the same time provides the full set of features of the original object interface implementation. New object interfaces that are expressed in thousands of lines of code can be implemented in approximately an order of magnitude less code [49]. While the use of Lua does not prevent deployment of malicious code, certain types of coding mistakes can be handled gracefully, and access policies are

used to limit access to trusted users [61]. We note that the use of a language like Lua allows us to more easily explore the space of programmability, but that the space is not defined in any way by a particular language.

4.2.2 Other storage systems

We use Ceph as our prototyping storage system. However, many storage systems contain a similar mix of sub-systems that are discussed at length in this thesis. For example, the distributed object store called OpenStack Swift [6] provides erasure-coding, replication, tiering, and durability features like scheduled data scrubbing. Unlike Ceph, Swift provides an eventual consistency storage model which expands the design space further when considering the scope of programmability in Swift.

Swift also contains aspects of programmability. For example, Swift Storlets are similar to the data interfaces that we will discuss in this chapter [90]. A Storlet is a function written by an application developer that can be run inside the storage system. Storlets depend on Docker for sandboxing and to provide an execution environment. The functions that are written as Storlets are stream-oriented, and have limited access to services provided by Swift. Unlike Storlets, as we will see data interfaces in Ceph provide a much richer set of interfaces to internal services and are not restricted to a stream-based model. Another key difference is that Swift provides only an object storage API, unlike Ceph which also has an associated scalable file system. And as we will see in Chapter 5 such a system is useful for building domain-specific interfaces.

The Hadoop distributed file system (HDFS) is a large-scale storage system that exposes a POSIX-like file interface to clients [121]. The system provides many sub-systems for erasure-coding, replication, and control over data co-location. In

addition, a Paxos-like system called Zookeeper is used to maintain distributed system-level metadata, and a scalable metadata service provides a hierarchical namespace to file system clients. Unlike Ceph, HDFS is not an object store and doesn't provide features for extending the interface to the low-level block abstraction that underlies file storage. However, some middleware layers that implement file formats utilize file system metadata to align application objects to block boundaries in a similar way that applications align data in an object.

The FastForward DOE project in preparation for Exascale systems produced a new non-POSIX object storage system called DAOS [77]. The DAOS system is one component in a larger storage strategy that includes both object-based storage and a flat namespace, along with a scalable POSIX file system provided by Lustre. The DAOS object interface allows applications to store both opaque binary blob data, as well as key-value data. In addition, DAOS provides a domain-specific interface for storing multi-dimensional data.

Pansas is an object-based storage system that includes a scalable POSIX file system that uses the underlying object-storage system for storing file data [84]. The Panasas system has a similar architecture to Ceph including object storage servers, and metadata servers. Pansas includes a Paxos-based cluster management system that is used to to replicate a configuration database [120].

Summary

Of the systems listed above, Swift Storlets are most similar to the data interfaces described in this chapter. However, Swift as well as all of the other systems above are listed to highlight that storage systems provide a wide array of internal sub-systems, and that this diversity is not unique to Ceph.

Throughout this thesis we will see how different application semantics are

mapped onto existing services found within Ceph. However, there is nothing inherently unique about the particular sub-systems found in Ceph—the mapping is dependent on both what is available in Ceph and what the application requires. The same is true of other systems, and each will contain a set of sub-systems that presents new challenges for mapping application semantics.

This observation further drives the need for declarative storage approaches described in Section 2.4. A simple reuse of a service in Ceph may be accomplished using a more complex mapping or an entirely different service found in a separate system. This challenge is inherently a portability challenge, and is what made POSIX and other standardized I/O interface popular and successful. Declarative approaches to programmable storage stand to provide a solution to this challenge, and will be necessary for the success of programmable storage.

4.3 Physical design

Developers of data management applications and services must often make important decisions about issues such as data layout that can have an enormous impact on performance. For example, it is common for database management systems to physically store relational data optimized for a particular access pattern that exploits properties of the underlying storage devices, such as storing data physically organized by rows, or by columns as is common for analytics database systems [58]. And certain data management operations in high-performance computing attempt to optimize I/O by reading and writing data that is aligned to physical or logical boundaries, such as storage media or advisory locks. We refer to the process by which these decisions are made as physical design, and the process itself generally covers how resources and interfaces are used and coordinated. Physical design is not limited to storage system developers; application develop-

ers often must understand aspects of the storage system to engineer effective and performant solutions. For example, file systems tend to expose a small number of configuration and tuning parameters, and offer generic high-level guidance such as optimizing for sequential I/O. Thus, application developers using a file system have a limited surface to explore optimizations, which has become a source of significant research and engineering to create application and middleware solutions that are able to perform well on a file abstraction. In contrast, programmable storage systems may expose a much larger scope of building blocks, making it easier for applications to directly express their desired requirements and semantics, but at the cost of increasing the size of the design space.

The programmability features found within Ceph expose three different interfaces for storing durable data. But as we will see, even with such a small number of interfaces the size of the design space can become very large. These three interfaces will be referred to in the remainder of this section as *omap* (short for object map), *byte stream*, and *xattr* (short for extended attributes). The *byte stream* interface provides a file-like interface for storing a sequence of bytes, and is efficient at storing and accessing large amounts of binary data. The *omap* interface provides access to a sorted key-value database that allows developers to associate arbitrary data with an object, and provides efficient random and range-based access methods. Finally, the *xattr* interface is similar to *omap* in that it provides access to sorted key-value pairs associated with an object, but has a different performance profile. As we introduce each category of application in this chapter, we'll discuss the use of these interfaces in detail. Note that these three interfaces provided by Ceph represent the current state of the system. As we discuss later in this section, it is an open question as to if these interfaces are sufficient, or if future applications may require or benefit from additional interface primitives.

Physical design is challenging for a number of reasons. First, the design space—that is the set of possible ways to build an interface—can often become very large. And without detailed knowledge of the system implementation, it can be difficult to reason about how to use existing resources and interfaces to reach a goal. Second, the internal interfaces exposed by Ceph and used by interface developers may be mapped onto a variety of implementations and hardware, potentially altering the performance profile of a design at runtime, as workload characteristics change, or at key life cycle events such as software upgrades, resulting in a spectrum of best design choices that are difficult to codify within a single implementation. One example of this is the recent introduction of very low latency non-volatile memories and the implications of that media on interface design. For example, Ceph imposes a non-trivial overhead in the fundamental use of an interface; that is, the CPU resource usage involved in executing an operation that traverses a large code path can be significant if a particular physical design forces a desired access pattern to invoke primitive methods with a high enough frequency. When new storage media are introduced with low enough latency, the CPU can then become a bottleneck itself. In this case a design may want to make use of more efficient APIs that reduce the number of invocations.

All of these challenges combine to expose a development experience that many developers may find to be too costly. In the remainder of this chapter we will examine how programmability can be used beneficially to build real-world applications that use resources exposed by a storage system in an application-specific manner. The primary goal is to chart a course for future work in programmability that will be based on declarative specifications.

4.4 Transactional data management

In this section we are going to examine one class of application that benefits from the transactional semantics of data interfaces in Ceph. The driving example that we have selected is a storage service that exposes a distributed shared-log interface. We use this driving example in this section to also provide a detailed account of the current state of building data interfaces, which expands on the motivational example from Section 2.3.1.

A common challenge in data management is maintaining consistency between two or more pieces of related data. For example, an index that provides efficient access to a large dataset along one dimension must be updated when the underlying data is changed. If the data and the index become inconsistent then an application can easily produce incorrect behavior by experiencing a false negative or positive result when querying for data.

Interfaces like the POSIX file interface provide little to no assistance to applications that wish to manage data with transactional semantics beyond coarse-grained solutions like POSIX file locking. This results in solutions found in data management systems like journaling updates, or multi-version concurrency control schemes. In many cases these workarounds introduce added or duplicated complexity, and in some instances inefficiencies such as double journaling writes are created [100]. Object-based storage systems may offer interfaces that can make certain types of data management simpler. For example, object-based storage systems often provide an interface for *replacing* an entire object in a way that readers do not see intermediate states. While this may be useful as it offers a transaction-like experience for updates, it forces updates to be coarse-grained and thus benefit only certain applications; far more flexibility is needed to support general applications. In contrast, Ceph provides finer-grained access by allowing atomic updates

to object data stored in any of its interfaces (i.e. omap, bytestream, and xattr), as well as the limited composition of these interfaces. However, all of these interfaces provide a fixed set of semantics that attempt to provide low-level primitives for a common set of application needs. That is, applications often require custom rules that govern how data is stored, read, and updated. When applications require higher level semantics, developers must translate between two or more sets of semantics, often using *workarounds* to achieve a particular outcome. These workarounds arise from a semantic gap between system interfaces, and is a source of difficulty in designing data intensive applications.

In the next section we will examine a simple example of how programmability can help to remove or make explicit these semantic translations and result in designs that are easier to reason about, and would otherwise be difficult or impossible to implement efficiently.

4.4.1 The CORFU abstraction

We have selected the CORFU log abstraction to demonstrate both the use of data interfaces for transactional data management, as well as to provide a detailed account of the current state of interface development in programmable storage systems. Recall from Section 2.2.2 that CORFU exposes a high-performance distributed shared-log interface providing strong consistency and a global ordering. The CORFU system is composed of three distinct pieces: a CORFU-specific storage interface, a high-level abstraction that maps log entries onto storage targets, and a network-attached counter called a sequencer that is used to assign clients log positions with high-throughput. Please refer to Section 2.2.2 for more details on the CORFU system, but we will repeat or introduce any salient aspects of this system in this in chapter.

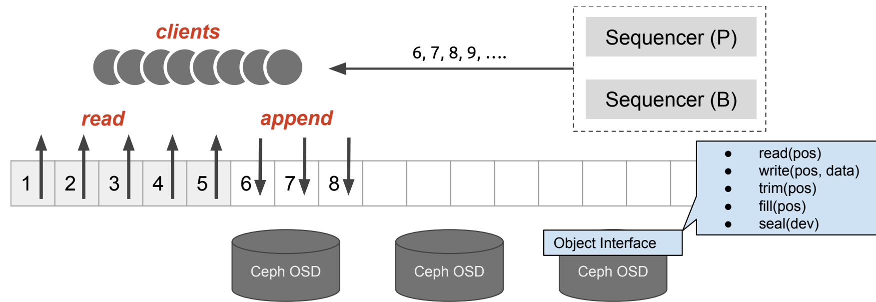


Figure 4.1: CORFU is mapped onto Ceph by making use of object interfaces that use programmability to reproduce the requirements of storage devices as defined by the CORFU protocol (e.g. positional r/w, trim, etc...).

Figure 4.1 shows a high-level architectural view of CORFU in which clients receive log positions from the sequencer, and dispatch I/O directly to storage devices. When creating a new storage interface like CORFU inside a programmable system like Ceph, one initial step is to identify the mapping between units of storage and I/O. The primary unit of storage in Ceph is an object, and Figure 4.1 shows the CORFU-specific object interface that is used in an analogous way to the custom SSD interfaces used in the original CORFU formulation. In the next section we will show how programmability is used to design such an interface.

4.4.2 Example design process

This section presents an example process of designing a data interface. We will be constructing the CORFU storage device interface, and will assume the role of a developer evaluating Ceph as a platform for implementing such a system. We assume a basic knowledge of the Ceph and CORFU architecture, as well as an operational understanding of how to build new object interfaces in Ceph, but lacking a detailed understanding of the implementation of Ceph and its detailed performance profile.

First we examine the basic requirements of the CORFU interface that we

want to replicate using programmability techniques. Then we present a high-level view of the architecture, and finally present a detailed account of the basic design process using programmability as it currently exists.

Requirements

One of the primary use cases for the CORFU log abstraction is to simplify the construction of scalable data management services, such as cloud-based metadata services, database management systems, and file systems [23, 17, 116]. The type of log that CORFU exposes is a strongly consistent shared-log that presents a global ordering to all clients. This type of log is typically implemented with solutions such as Paxos [75], but achieving high-performance with these solutions is difficult due to natural bottlenecks in the architecture (e.g. funneling writes through a single node). The types of systems that CORFU targets such as cloud-scale data management systems may require both high-throughput and low-latency I/O. The CORFU abstraction solves one key aspect of providing a high-performance log through its fundamental architecture which decouples the assignment of log orderings from I/O. However, an implementation of the CORFU interface must still make use of hardware capable of delivering high performance. For example, the original CORFU design mapped log entries onto a cluster of flash devices, but it is the design of CORFU itself that allows the system to exploit the high-performance nature of a cluster of fast media.

One requirement of building a CORFU log abstraction using Ceph programmability is that the architecture that provides CORFU with the ability to achieve high-performance is retained. However, Ceph itself today as a prototyping platform does not provide the same performance as a cluster of network attached flash devices. By recreating the same optimization techniques, however, we take

advantage transparently of enhancements to Ceph as they become available. For example, recently there has been increased interest in running Ceph on flash-only clusters and work is currently being done to increase the performance of Ceph for next-generation non-volatile memories. This transparency means that a log can be *configured* to be stored on fast devices, or slow devices like spinning media to optimize for capacity. And as the system is able to more fully exploit the performance of new devices, so to will an interface built on Ceph such as the one we will be exploring.

As a general service, a log can also be used in more traditional modes such as a write-ahead log. In our discussion with industry, we have found that optimizing for tail latency and read throughput for large sequential I/O needed for log scans (as opposed to fine-grained random reads) is an important operational mode [94]. As we will see these metrics can be taken into account during the design process of data interfaces.

High-level solution

The creators of the CORFU abstraction designed the system to achieve high-throughput by striping log entries across a cluster of flash devices providing scalable I/O through parallelism. A high-level question that must be answered is how this same parallelism will be achieved in a mapping of the protocol onto Ceph. Recall from Section 4.4.1 that objects in the Ceph storage system are mapped onto physical storage devices, and can be used as an approximation of a unit of I/O parallelism. We use this knowledge of the underlying storage system to arrive at a high-level design in which log entries are striped across a set of objects (e.g. using round-robin addressing).

> **Challenge.** Translating optimization techniques onto the target programmable storage system requires understanding fundamental as-

pects of its design, such as understanding how I/O may be parallelized. However, this issue goes deeper; for example, in addition to physical parallelism in Ceph being achieved through the use of objects as a basic unit of I/O, logical constraints such as locking also exist. Typical configurations hide issues such as locking, but designing an interface with a misconfigured system could produce suboptimal solutions that become codified.

The data interface expected by the CORFU protocol operates on log entries addressed by their global log position. This has implications on the design because the implementation of a data interface must be capable of addressing a wide range of potential log addresses using techniques such as indexing. The log entries managed by the CORFU storage device specification places log entries into one of three states: *free*, *written*, and *invalid*. This means that the data interface must be able to manage at least 2 bits of metadata per log entry, along with the opaque log entry data itself. The exact set of metadata required ultimately depends on the full set of semantics of the interface which may be adjusted by a particular implementation. For example, CORFU is able to use compact data structures to represent indexing information by assuming fixed size entries. However, one useful deviation from the original CORFU design may be to support the storage of variable length entries. If this is the case, then in addition to basic state information, per log entry metadata must also include details such as the size of a particular entry. When these details are important to the design discussion we will highlight them.

In the remainder of this section we detail a sample process of physical design for the CORFU interface in Ceph. We will look only at the *write* interface of the CORFU storage abstraction that is used for storing new log entries. Other interfaces such as *trim* and *read* present similar design challenges, but as we will see focusing only on *write* is more than sufficient to support our claims and demonstrate storage programmability. In fact, the multiplicative affect on the

design space of simultaneously attempting to optimize for two interfaces would result in a discussion that would be difficult to present.

Log entry storage strategy

The current state of building interfaces in programmable storage systems can require an arduous process of trial and error. This is especially true for developers that do not have an in-depth knowledge of the performance profiles of the internal storage interfaces. In this section we will present a window onto the design process of one interface that highlights the trials that developers are subjected to when exploring the design space of new storage interfaces.

As discussed in the previous section, Ceph exports three internal storage services for building new data interfaces. These three are the bytestream, object map (omap), and extended attributes (xattr). The CORFU interface that we are constructing will store persistent log entry data, and must make use of these interfaces for data management. In addition to the log entry data, each entry requires that a small amount of metadata must also be managed. This means that there are nine possible combinations of storing entry data and metadata using the three provided storage interfaces.

> **Challenge.** It is an open question if the primitive interfaces provided by Ceph (e.g. omap, bytestream, xattr) are sufficient for all applications. They are useful for a broad range of applications, but other interfaces (e.g. mmap) may offer advantages for certain applications. In this respect the type of analysis we are performing will never be finished. Hiding this evolution from interface developers will help adoption of programmability techniques.

Rather than implementing all nine strategies, it would serve us well to eliminate some possibilities. To do this we can conduct a simple experiment to examine the performance profile of writing log entries into each of these interfaces, and draw conclusions from their relative performance across a range of entry sizes.

Figure 4.2 shows the throughput in appends per second when writing 128 byte entries into each of the three storage interfaces over a ten minute window.

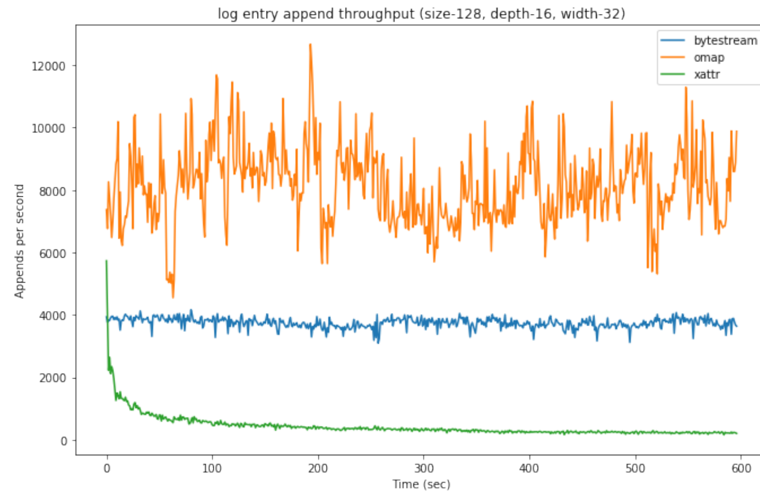


Figure 4.2: The throughput in operations per second of writing 128 byte entries into each of the three storage interface provided natively by Ceph.

The experiment was run using one Ceph storage node configured with a modern Intel CPU with 20 cores, 120 GB of RAM, and an enterprise grade SSD used as the backing store for Ceph running with Bluestore. We ran the Ceph monitoring daemons on a separate node, and generated our workload on the OSD to help eliminate network jitter from the results. The first immediate conclusion that we can draw from this graph is that using extended attributes to store entry data is not viable; in fact the performance degrades over time.

Understanding why the extended attribute interface has this performance profile is not necessary to rule it out as a candidate for storing entry data. However, understanding its implementation is useful for understanding the conditions in which it may be used successfully. Extended attributes are serialized by Ceph in a way that causes them to accumulate in size. Thus even though our experiment is storing small, 128 byte records, the total accumulated size becomes larger with each operation. Later in this section we will revisit extended attributes and how

they can be used effectively.

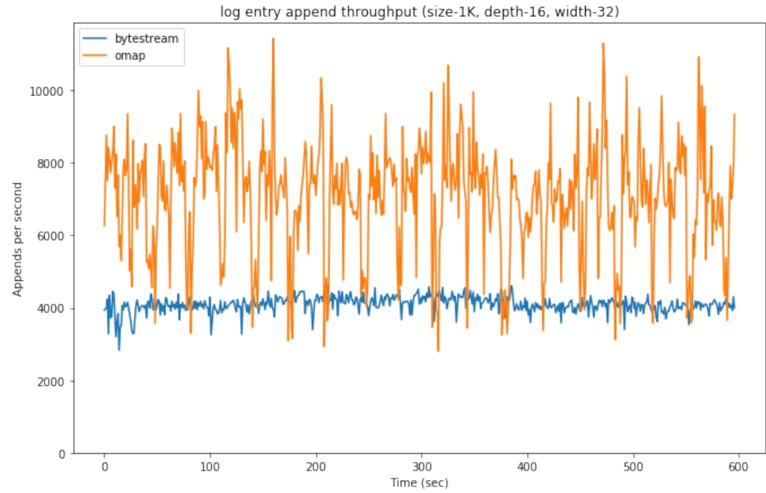


Figure 4.3: The throughput in operations per second of writing 1KB entries into each of the three storage interface provided natively by Ceph. As the size of the entry being stored increases, the variance in performance of the omap interface also increases.

The second conclusion that we can draw from Figure 4.2 is that the omap interface provides far better throughput than the bytestream interface when storing 128 byte entries; nearly a 2x improvement. Without any further experiments, selecting omap as the storage interface for entry data would provide the highest log append rate. This trend continues as the entry size increases, as shown in Figure 4.3 in which 1KB entries are written, where the omap interface continues to provide better throughput than the bytestream interface. The figure also indicates that the performance of the omap interface exhibits higher variance, which we will discuss shortly. Also note that the extended attribute interface has been removed as a storage option after discovering its performance profile, discussed in the previous experiment.

We repeated this same basic experiment for a wide range of entry sizes, and summarize the average throughput in Figure 4.4. The x-axis shows the size of the entries being written, and those labeled `X+1` correspond to `X+1` bytes (e.g.

4096 and 4097 bytes). As shown in the figure, the omap interface can provide far better throughput than the bytestream interface when storing log entries that are less than 4KB in size. For entry sizes larger than 4KB, the bytestream interface outperforms the omap interface, except for entry sizes of 4KB+1 bytes, and the two interfaces perform virtually the same with 8KB+1 sized entries.

The reason that 4KB entry sizes are an exception has to do with both an implementation detail within Ceph, and a configuration parameter that controls when I/Os trigger allocation and read-modify-write activity. For example, in our experiment we have set this parameter to 4KB, and in fact its default value is 16KB, which also varies depending on the type of media being used (e.g. disk vs SSD). While small writes and unaligned writes (e.g. 4KB+1) achieve better performance with omap, the overhead of the omap interface begins to dominate as the entry size increases at which point the bytestream interface provides better throughput.

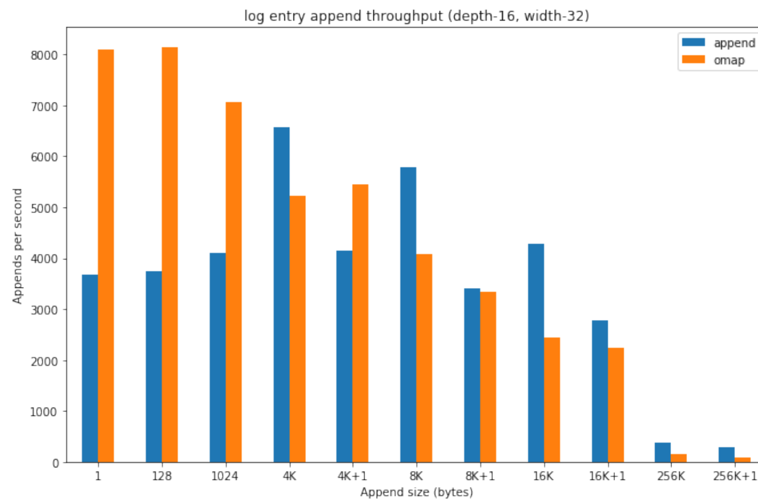


Figure 4.4: The throughput in operations per second for storing log entries of varying size in either the bytestream or omap interfaces.

Unfortunately this example only highlights a portion of the complexity. For example, the exact cross-over points for performance will depend on both me-

dia and configuration parameters. For instance this experiment uses a custom storage backend built specifically for Ceph called Bluestore, which itself depends on RocksDB. Another common backend used in production runs on top of a file system like XFS, and uses the kernel buffer cache. Furthermore, recall from Section 2.3.2 the unwieldy number of configuration parameters and tunable settings that exist in Ceph, in addition to the ability to configure the system by swapping out critical components such as the storage engine with alternative implementations. Invariably these factors combine to produce a system that can be configured in multiple ways to produce different performance profiles.

As previously mentioned, other metrics such as tail latency can be an important factor when designing an interface like a high-performance log. For each of the experiments that we ran, we also collected a latency distribution of all I/O operations. For entry sizes less than 8K, omap provides better median latency by a factor of 1.2x to 2.2x. However, at the 99th percentile, the bytestream interface provides better tail latency of between 4x and 11x for entry sizes larger than 1K bytes. This result turns out to be unsurprising when considering the implementation of these interfaces. The omap interface is implemented using RocksDB, and this particular storage engine is well known for not performing well when storing large values due to write amplification [79]. On the other hand, the bytestream interface can excel at large I/Os because it issues raw I/Os directly to the underlying block device.

> **Challenge.** Building a new data interface is effectively a multi-dimensional optimization problem. An implementation strategy is dependent on both the workload characteristics (e.g. the distribution of I/O sizes), high-level performance goals of the interfaces, as well as the system configuration and performance profile of primitive interfaces. Furthermore, optimizing for certain metrics (e.g. tail latency) may mean making large sacrifices in the performance of other metrics.

Thus far our design process has focused on the bare minimum operation of

storing log entries, but has been very useful in identifying a high-level performance profile for the construction of the CORFU domain-specific interface. Next we will consider how metadata should be treated by our implementation.

Metadata management strategy

As we have seen the bytestream and omap interfaces provide differing performance profiles when storing log entries, and the best choice is not always clear. However, we have yet to consider how metadata should be handled in our implementation. There are three states that each log entry can be in (unwritten, written, invalid) which thus requires a minimum of two bits to represent. Depending on the exact implementation semantics (e.g. fixed entry sizes, maximum entry sizes, dynamic sizes) an implementation may require a small number of additional bytes to encode information like bytestream offset (if the bytestream interface is used), and entry length.

For each of the two interfaces, omap and bytestream, we can select between either as well for metadata storage yielding four design possibilities. We have omitted consideration for now of extended attributes, and revisit this interface at the end of this section. We will repeat a similar process to winnow down our design space. First let us consider the case of storing log entry data using the omap interface. Based on the results shown in Figure 4.4 we can identify the conditions under which using omap to store log entries is beneficial, but we must also take into account the additional metadata storage.

These results show that omap performs well compared to the bytestream interface for entry sizes smaller than 4KB, but when evaluating where to store a few bytes of metadata per log entry, we can see that even very small writes to the bytestream (e.g. 1 byte) result in poor performance. Thus, when storing log

entry data in omap, combining the overhead of metadata storage into omap as well will offer the best performance. For example, each entry in omap may store its associated metadata as a prefix on the entry data in the value associated with an omap entry, provided the total size falls within a range that is optimal for omap.

However, consider the case of writing an aligned I/O to the bytestream. Using an approach that combines the metadata with the log entry data (e.g. as a header) would result in an aligned I/O that is turned into an unaligned I/O, perhaps altering the decision about which interface to use. For example, when storing a 4KB log entry one would achieve higher throughput storing the entry and the metadata in omap compared to writing the 4KB entry plus a metadata header to the bytestream (as shown by the results of writing 4KB+1 byte entries) even though a 4KB write to the bytestream outperforms a 4KB write to omap. A corollary to these results is that padding a small I/O to achieve an aligned write to avoid using the omap interface can provide performance benefits. Finally, even though the throughput of an unaligned bytestream write can be worse than using omap, a strict priority on tail latency may dictate using the bytestream interface despite the lower throughput.

The final approach to managing metadata that we will examine is one in which omap is used to store metadata, and the bytestream is used to store log entries. In this approach every write becomes a write to both the omap data interface and the bytestream interface, and the atomic execution semantics provided by Ceph ensure that the two data locations remain consistent. The results of this approach are shown in Figure 4.5. The first thing to notice is that for log entry sizes less than 4KB, and the unaligned write of 4KB+1, the hybrid approach performs poorly compared to using omap because of the lower performance of

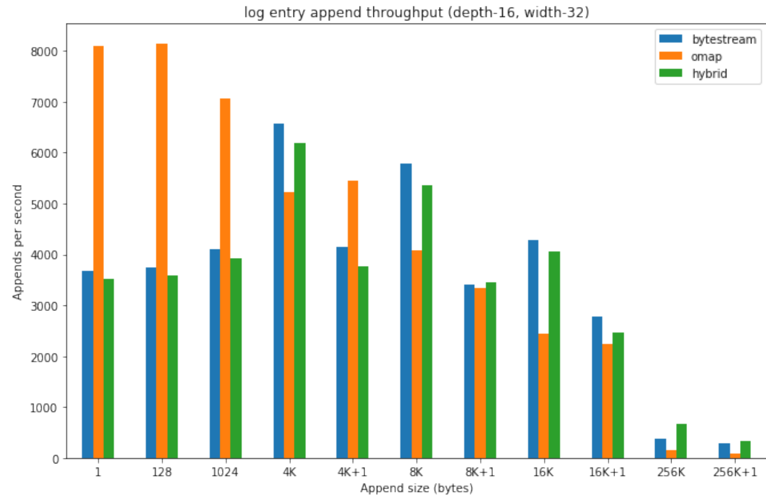


Figure 4.5: The throughput of writing log entries using three strategies: omap, bytestream, and a hybrid in which log metadata is stored in omap, and log entry data is stored in the bytestream.

writing small, unaligned entries to the bytestream. This would appear to be an uninteresting result, except for the following observation: in each of these cases, the omap interface is already capable of outperforming the bytestream interface according the cost model that we have observed through experimentation. Thus, in these cases the unified approach can be adapted to store the entires in omap which is already being used for metadata when the model suggests a performance improvement.

Likewise, the same model can be used to encode rules for when to redirect writes to the bytestream. Thus, a unified approach to managing metadata in this way allows an adaptive solution, that always outperforms the alternative.

> **Challenge.** Collecting data that exposes a performance model is only one step in the design process. Acting on this model requires techniques found in database systems that use models and the current system state to select the best execution strategy. Complex logic required for an implementation to handle all of these cases (and this is only a selection) must be written, and ultimately rewritten as assumptions change without a different approach to building interfaces.

Extended attributes interface

Throughout our discussion of the design process we have ignored the extended attributes interface after observing that its performance degrades as more data is written into it. This held for both storing log entry data, as well as per-entry metadata that grows as more log entries are written (albeit at a slower rate).

The extended attributes interface is nearly identical to omap in that it provides an interface for storing sorted key-value pairs, but the xattr interface provides very fast, low-latency read access because the data is always cached in-memory. Data stored as extended attributes are serialized and stored along with an internal object metadata structure that is saved every time the object is updated. Therefore, we expect the overhead of using the xattr interface to be less than the omap interfaces provided the total size of data managed by the xattr interface falls below a threshold.

In order to be able to use such an interface one needs to be able to compactly represent entry metadata. If we assume a restricted use case with fixed log entry sizes, then we can make use of a calculated placement strategy and represent log metadata by using a bitmap that encodes the state of each log entry. Raw bitmaps can reduce space usage by a factor of 8x, but even storing a modest number (e.g. 10,000) of entries per object, two bitmaps would still require over 2KB of space that must be decoded, re-serialized, and written for each log write operation.

A class of bitmaps called compressed bitmaps can represent compact bit patterns efficiently using techniques such as run-length encoding. Figure 4.6 shows the serialized size of a state of the art compressed bitmap [32] of size 10,000 when the bits are set in a random order. When the bitmap is small a few sparse bits still enable good run-length compression. As the number of bits grows the sparseness of the bits degrades the effectiveness of the compression technique. Techniques

like run-length encoding operate efficiently when there are repeating patterns (e.g. a long series of set bits). A random workload is thus an adversarial workload that largely minimizes the benefits of techniques like run-length encoding. After some period of time a majority of the bits are set, and as larger groups of bits are set, run-length encoding again becomes effective. While the peak data usage should be a concern there are two things to consider. First, a nice property of the compressed bitmap is that if an adversarial case occurred, there is the opportunity for the size to eventually decrease as more log entries are written and the density of bits that are set increases.

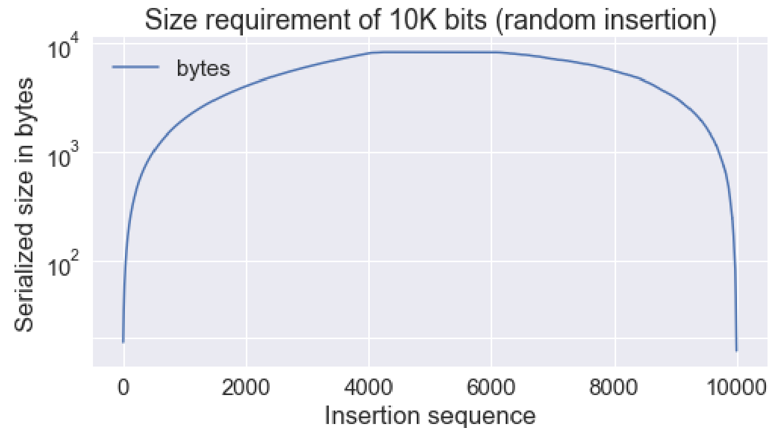


Figure 4.6: Simulated worst-case serialized bitmap size with a random arrival order of bits being set. 10K bits.

The second things to note is that fortunately, a workload in which all log appends arrive in a random order is *highly unlikely*. Consider a log configured to store 10,000 entries per object, with the log striped across 25 objects. Since the sequencer in a CORFU implementation makes log assignments sequentially to clients, in order to even create the conditions for such a worst case arrival order to occur there would need to be around 250,000 inflight I/Os. A workload pattern that is more reasonable to expect is one in which the object is filling up primarily sequentially (the best case for a compressed bitmap), but reorderings occur within

a bounded window at the frontier of the object append position.

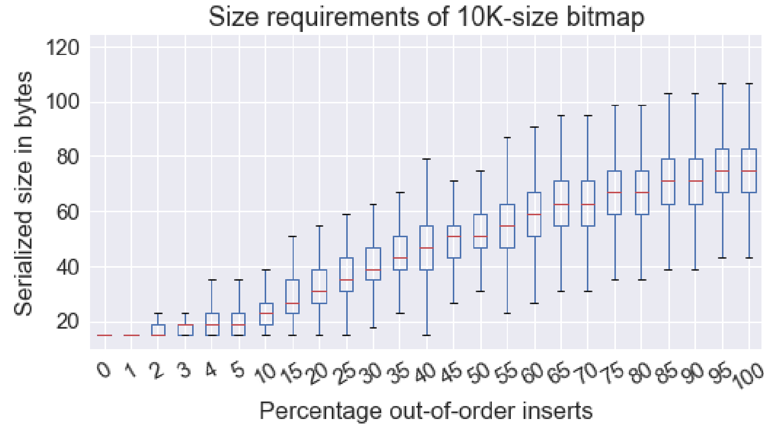


Figure 4.7: Simulated partial arrival order. Arrivals are out of order, but bounded by a sliding window across the total set of log positions.

We simulated this arrival model by reordering a variable set of bits selected randomly from the total 10,000 and constrained the reordering to occur with a small window of 100 bits centered on the selected position. Figure 4.7 shows the results of this experiment. The x-axis shows the percentage of total bits that are reordered within their bounded window. For example, when 10% of the bits are reordered, the median size of the bitmap over the course of the entire experiment is approximately 25 bytes, a significant improvement in real-world compression ratio compared to the worst case index size observed for an adversarial case.

We found that even when storing as little as 20 bytes of a simulated index in the extended attributes interface that there was no real performance benefit. Considering our knowledge of the implementation suggesting there may be a performance benefit when storing a small amount of data using xattr over a different interface such as omap, this result is counter-intuitive when considering the alternatives that stores thousands of individual metadata entries. Without an extensive analysis or ability to contact experts, usage of extended attributes may remain fully unexplored as an option for a developer of a new data interface.

Challenge. Techniques such as utilizing compressed bitmap indexes, or padding log entry I/O to achieve aligned writes, are engineering solutions that provide specific optimizations. In a system allowing interfaces to be specified declaratively these engineering techniques must be generalized and exposed to the optimizer as tools for navigating the design space based on query semantics and available cost models. Integrating new techniques will be an on-going process.

4.4.3 Summary

We have built many data interfaces, and the process outlined in this section for constructing the CORFU interface reflects a common process that involves first collecting basic requirements, and evaluating different high-level approaches to winnow down the design space. Overtime we have developed intuition about what works well in different situations, but this intuition is difficult to communicate and it is subject to immediate invalidation as the system, workload, and configuration evolves.

However, the benefits of building such interfaces has remained clear. It may be the case that application developers (ourselves included) can bare the cost of evaluating designs and arriving at a solution that works well. Unfortunately, as we saw in Section 2.3.1 even a simple system software upgrade can have dramatic effects on design decisions. These types of unforeseen events are difficult to account for, and can have far reaching consequences when interface changes are required. The design space becomes even more complex if interfaces can benefit from new primitive services and costs must be weighed with respect to migrating to new formats and data management strategies that work with existing data.

4.4.4 Declarative approaches

Declarative specifications allow developers to decouple what they want to achieve from the mechanisms that provide those results. In this section we showed

that simple cost models can be built to guide how data interface should behave. By all accounts the sophistication of state-of-the-art database optimization strategy covers the necessary requirements of the decisions we have outlined in our evaluation. In Section 2.3.1 we highlighted the use of the Bloom language to encode the semantics of interfaces such as CORFU that we have explored in this section as well. The Bloom language in particular is built upon a formal foundation that allows the full breadth of database optimization research to be applied.

However, database query optimization and planning is not a magic wand, and is only as good as the cost models and tools that are available. For example, we showed that small log entries could be padded in order to create an aligned I/O that performs better. This type of strategy must be *taught* or made available to the set of techniques known to the optimizer. Ceph doesn't expose a bitmap index interface, but as we saw this can be a useful tool. An approach to programmability using a declarative specification would need to also be aware of tools such as indexing strategies like bitmaps that it can make use of. A storage system may even contain sub-systems that have not yet been exposed for reuse by the system when producing an execution plan for a declarative specification. For example, if Ceph used a bitmap index internally this could later be exposed for reuse, potentially providing additional benefits.

4.5 Computational resources

In this section we will examine how programmability is used to build storage interfaces that expose system resources such as CPU, memory, and I/O bandwidth. Unlike the previous section that examined how a distributed shared-log interface benefits from fine-grained transactional data management, in this section we will look at a large-scale database management use case that utilizes storage system

resources for accelerating query execution performance by offloading computation and indexing services to the storage system.

Scalable database services (e.g. cloud-based SaaS offerings) rely on efficient scalability of both database processing and storage. Ideally each resource may scale independently, but in practice there are often trade-offs that are made. In the next section we describe two common techniques for scaling database systems, both of which have significant limitations when being considered for cloud-based environments that rely on flexible scaling. These limitations are used as motivation for the development of a database system architecture called Skyhook [38] that decouples storage and database processing, allowing each component to scale independently.

Skyhook was designed from the beginning to exploit storage system programmability by aligning the logical boundaries of database abstractions (e.g. relational tables) to low-level object storage abstractions found in common cloud-based storage systems such as Ceph. This approach supports an architecture in which offloading common query optimizations such as selection and projection is simplified because storage and computation are aligned at a fine-grained level.

In Section 4.5.3 we present results of this architecture by showing the benefits of using programmable storage below a PostgreSQL database node running common TPC-H queries [105]. We show that the architecture can increase performance by reducing data transfers, exploiting local CPU and I/O bandwidth within the storage system, and utilizing internal sub-systems such as indexing.

4.5.1 Background

Figure 4.8a shows the high-level architecture of a standard single-node database system (e.g. MySQL or PostgreSQL). In this architecture resources local to the

database node (e.g. CPU and memory) are used by the database system which has access to attached storage (e.g. SCSI or NFS). Scaling a single-node database system has many limitations. For instance, a scale-up approach adds additional resources to the database node, but this has physical limitations, and while scaling out storage can increase capacity and throughput, these are also physically limited by the network or local bus.

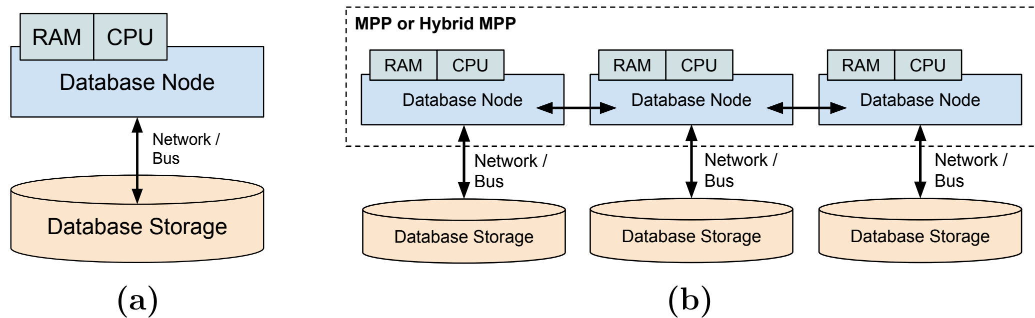


Figure 4.8: (a) The scalability of single-node database systems are physically limited. (b) Common scale-out approaches have operational limitations that make scalability inflexible due to storage affinity and static resource allocation.

The standard approach to horizontally scaling a database system is to use a massively parallel processing (MPP) architecture in which the resources of many nodes are tightly coupled to create a unified database system (see Figure 4.8b). A similar approach referred to as a hybrid-MPP architecture forms a logical database system by coordinating multiple independent database instances. Both of these architectures have proven to be scalable approaches to building database systems. However, there are limitations to these approaches in a cloud-based environment where flexible scalability is needed.

These limitations arise from common methods by which storage is managed. First, both system architectures tend to create a strong affinity between a node and storage that it may address (e.g. only data in a locally attached disk). When a system grows, shrinks, or otherwise is reconfigured in a way that existing partitions

are changed, data shuffling occurs and can be a stop-the-world data management event. This storage affinity also limits the flexibility of utilizing CPU and memory resources. Because database nodes take ownership of partitions of the database, processing a partition is limited to a subset of the total compute resources of the cluster, even when capacity is otherwise available.

Next we discuss the salient features of the Skyhook architecture which address these limitations, show how Skyhook makes use of programmable storage systems, and present results from several experiments.

4.5.2 Skyhook

The Skyhook database architecture is an experimental system that seeks to address limitations in existing scale-out database systems by decoupling data management concerns into independently scalable components. Skyhook abandons storage abstractions such as POSIX files for more flexible object-based storage. Figure 4.9 shows a high-level view of data organization in Skyhook in which a relational table managed by a database is partitioned into a number of shards that are stored in objects within a Ceph cluster. Skyhook ensure that these partitions are constructed below high-level table abstractions managed by a database node, and as a result, data rebalancing is handled transparently by the storage system, and only logical repartitioning results in global data shuffling (e.g. sharding a table on a new key).

While systems such as Ceph provide flexible scale-out storage, it is the use of the object abstraction and the programmability of the system that allows Skyhook to use a single database node to accelerate many common query processing operations. Critically the object-based abstraction allows Skyhook to precisely align relational table partitions with the units of I/O and CPU parallelism in

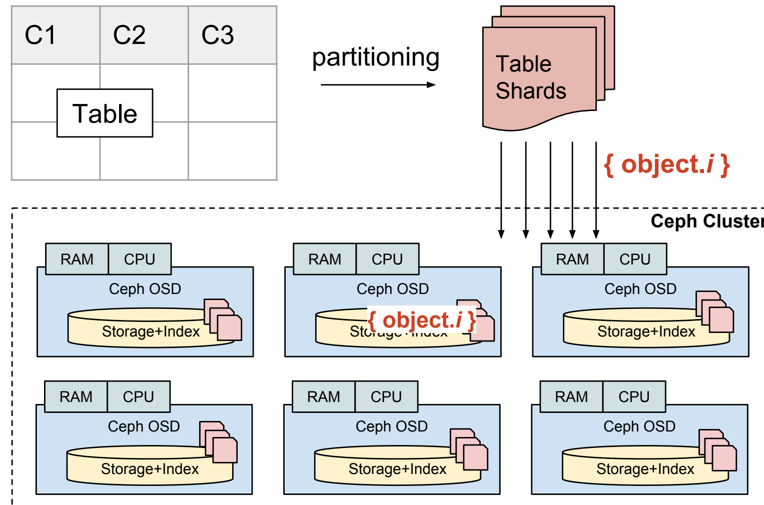


Figure 4.9: Skyhook stores relational tables by partitioning the table into shards and storing each shard in an object within the Ceph storage system. This allows transparent rebalancing to Skyhook.

the storage cluster. This supports a common method for query acceleration: the use of push-down processing, such as running selection and projection operations closer to the data.

Push-down processing is a technique found in some database systems that reduces the amount of data that must be examined and transformed in order to answer a query. For example, a database executing a query that selects 10% of the rows in a table by scanning will read and materialize all rows only to retain a fraction of the total data set size. Push-down processing can be used to eliminate rows from consideration early on in the process to reduce the total amount of work, and data transferred. Other forms of push-down processing include computing aggregates (e.g. sum, mean, max) using resources closer to the storage. When storage and compute are co-located, push-down processing can exploit I/O and CPU parallelism for query evaluation such as reading and processing rows concurrently.

Database systems that use existing storage abstractions such as POSIX files do

not have a straight forward mechanism for exploiting storage system resources. Interfaces such as a bytestream only expose read and write interfaces, and the implementation of these interfaces have no knowledge of the actual structure of the data being stored (e.g. rows and columns). Some systems layer solutions over the bytestream interface creating an intermediate abstraction that achieves a similar result. However, these systems rely upon affinity mappings that create static resource allocations that are not flexible, and may duplicate common functionality and services. Other systems like Hadoop address this issue by co-locating distinct storage and compute systems, and use locality information and resource scheduling to achieve locality goals.

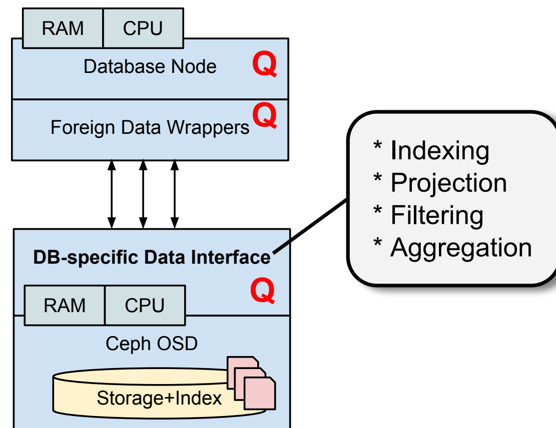


Figure 4.10: Skyhook is structured as a standard single-node database connected to a scale-out storage cluster such as Ceph. The database node connects to each storage server through a custom database-specific interface that allows the database to use storage system resources such as the CPU, and indexing services to accelerate query performance.

In contrast, Skyhook takes advantage of storage system programmability by mapping both read and compute nodes in a query plan to objects within the storage system, and using a custom data interface to implement methods for query acceleration. Since objects are well-aligned to CPU and I/O resources, fine-grained resource utilization is far easier to achieve. Figure 4.10 shows the

interface between the Skyhook database node and the underlying storage system. Each Ceph OSD exposes a custom object interface through which the database interacts with table partitions. The interface understands both the structure of the data, as well as common operators used in queries such as indexing, projection, and aggregation. Programmability is flexible enough to support more complex processing operations such as partial aggregates, pattern matching, sampling, or customized processing such as domain-specific user-defined functions.

4.5.3 Results

We have constructed a storage interface using Ceph programmability features that allow a database system to push down query operators (e.g. selection and projection). We use four queries that contain a subset of the common operations found in the TPC-H benchmark suite. Our data set consists of approximately 140 GB, and contains 1 billion rows. The relational table is partitioned into 10,000 objects each roughly 4 MB in size. We have restricted our experiments to a fixed set of queries running on a static data set. The scenario reflects a common process in which a large dataset is loaded in bulk into a database after which queries may be run against the data set. The queries were evaluated on a Ceph cluster, and we repeated the experiment using between 1 and 16 OSDs each containing 20 CPU cores and approximately 120 GB of RAM.

Queries that contain operations that may be run in the storage system are applied to each of the 10,000 objects and metrics such as runtime and local resource utilization are collected. Unless otherwise noted, results are the median value of three runs of an experiment. The database node manages the parallel execution of these push-down operations using a queue depth of 12 operations per Ceph storage node. We found that this configuration provided good performance and

demonstrated the scalability gains of exploiting remote resources.

Data reduction

We demonstrate how the database system can accelerate query execution by reducing the amount of data that must be returned and processed by the database node using two queries. The first query simulates a standard selection query that returns all rows that match a predicate. The equivalent SQL is given by `SELECT * FROM lineitem WHERE extendedprice > 71000`, and the query parameters are selected so that 10% of the rows match. Figure 4.11 shows the result as we vary the number of storage nodes used. The result labeled *client-side* is the cost to return all of the data to the database node and process it locally, while the *server-side* result executes the filtering operation inside the storage system and returns the rows that match.

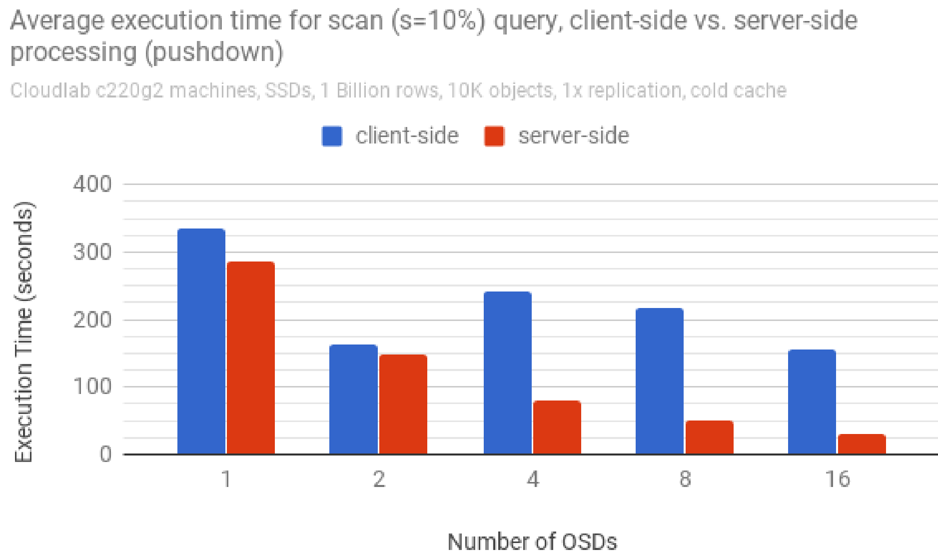


Figure 4.11: Runtime of a 10% selectivity query executed on a single client compared to using custom object interfaces and push-down predicate evaluation.

In this experiment the result labeled *client-side* represents the case in which

the database node is responsible for fully evaluating the query, and reading the data stored in the storage cluster remotely. In this configuration the database node always does the same amount of total work, however the performance improves with more OSDs because of the increased I/O throughput that is available for scanning the dataset. The result labeled *server-side* corresponds to a configuration that is identical to the *client-side* configuration except that the filtering predicate is applied on each OSD to the 10,000 objects. The reduction in runtime is three-fold: first, like the *client-side* case the system benefits from increased I/O throughput. Second, by utilizing the total set of CPUs available in the storage cluster to perform filtering the amount of data returned to the database node, and by extension the amount of data transferred over the network is reduced by 90%. And third, the amount of work that must be performed by the database node itself is reduced to only those rows that match the predicate.

Compute

While reducing the data transferred is a valuable optimization, queries with simple predicates over primitive types (e.g. numeric comparison) as we saw in the previous example, do not tax the compute resources. Figure 4.12 shows the results of running a query that uses a regular expression predicate that also selects only 10% of the total set of rows. The query is equivalent to the SQL statement `SELECT * FROM lineitem WHERE comment iLIKE '%uriously%'`.

The results are similar to the data reduction experiment in the previous section, shown in Figure 4.11. In this case the nodes we use (as well as the client node) are able to absorb the additional cost of evaluating a regular expression.

Figure 4.12: Runtime of a 10% selectivity regular expression query executed on a single client compared to using custom object interfaces and push-down predicate evaluation.

Index acceleration

Finally we highlight the use of internal indexing services found in the storage system for accelerating point queries against the database. Recall that the dataset we are using is a table with 1 billion rows partitioned into 10,000 objects. The query we evaluate is a point query with predicates evaluated against two columns. The equivalent SQL statement is given by `SELECT extendedprice FROM lineitem WHERE orderkey=5 AND linenumber=3`. In order to accelerate this query we have built a data interface that is used to construct a compound index on the *orderkey* and *linenumber* columns. This index is stored in the internal indexing service provided by the omap indexing interface provided by Ceph.

The compound index maps the attributes of each table row onto the bytestream within the object, providing the offset of the row associated with the index entry. This allow a query to avoid scanning the data set in the object by looking up the

exact row location using an efficient indexing mechanism. This index is built and maintained on a per-object basis after the initial dataset is bulk loaded into the system using the omap indexing service interface.

While this indexing strategy provides accelerated point queries for the rows contained in an object, the set of objects that must be queried (i.e. the set of objects a row *may* be found in) is dependent on the data partitioning strategy, and natural ordering of the rows. For example, if there is no relationship between the partitioning of data and the columns being queried, then all partitions must be queried. This is the case for the partitioning and the query that we evaluate. Figure 4.13 shows the cost of performing the point query using client-side and server-side scanning, as well as the index accelerated lookup that queries all 10,000 partitions. The results show that the indexing service can provide significant performance benefits for query execution.

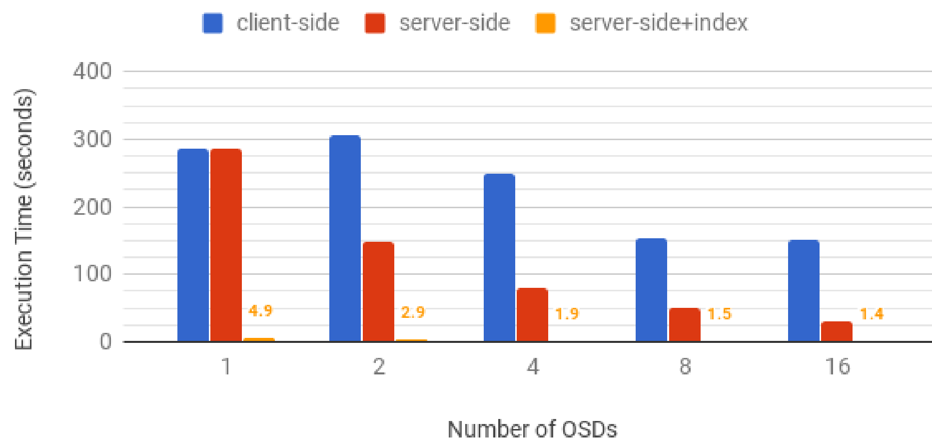


Figure 4.13: The runtime of evaluating a point query over 10,000 objects by scanning at the client, scanning at the server, and using an index over all rows.

What is fascinating is that by utilizing the full I/O bandwidth and CPU re-

sources of the cluster, a query operation over 1 billion rows can be accelerated by taking advantage of the fine-grained data and index partitions managed by the system even when 10,000 remote operations are invoked.

Index construction

Finally, we evaluate the performance of building the index over the entire table. The process happens in two steps. First, the raw data is loaded, and then a second pass is made to build the index. These two steps could be combined, but we have separated them to help break down the costs involved. Figure 4.14 shows the runtime costs of these steps as we vary the number of objects that the table is partitioned into, corresponding to larger or smaller table partitions. In each experiment the dataset size remains constant. The runtime cost of storing the objects increases as the number of objects increase because the per-object overhead costs accumulate. These costs consist of overheads such as network round trips, and the latency involved in establishing an object context on an OSD.

Interestingly the runtime costs of building the index decrease as the table is partitioned into smaller shards. The reason for this reduction is a result of the efficiency of inserting key-value pairs into the omap database, and the number of rows being indexed per object. The implementation of the method that builds the index inserts all index entries for an object in a single object transaction. Thus, when the table is partitioned into fewer objects the size of the resulting transaction is larger. However, the internal implementation of transactions in Ceph is inefficient for very large omap bulk inserts due to issues such as the serialization and deserialization of in-memory data structures. A larger number of objects results in smaller partitions, and thus more fine-grained omap updates which are more efficient for the system to handle.

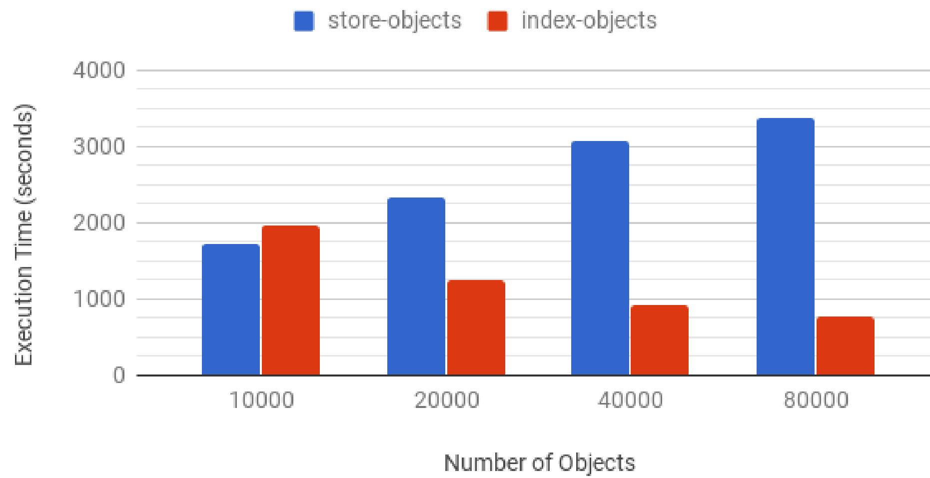


Figure 4.14: The runtime cost of storing and indexing table partitions as a function of the number of partitions.

This result highlights an inherit challenge in building domain-specific interfaces of developing with a changing set of primitives and a dynamic cost model. For example, a developer may reasonably expect that bulk inserting larger partitions would be more efficient, but internal details negate this assumption. However, an implementation that handles larger partitions by inserting smaller batches within the same transaction may outperform the result shown with smaller partitions because it also avoids the network round trips. While the omap interface in Ceph was likely not designed with such a use case in mind, the internal optimizations that could make bulk inserts more efficient would be a reasonable future enhancement. If a developer builds an interface and uses configurations that optimize for larger numbers of objects, then a more efficient interface may not be easily taken advantage of because the previous cost model was codified into a static design.

4.5.4 Summary

This section examined programmability in the context of a database management system. The results show that programmability can be used to build data interfaces that are viable and useful for applications that can benefit from storage resources like CPU, memory, and indexing services. A generalization of the challenges presented when developing the log-based data interface in the previous section also apply to the database management use case, though the exact resources and cost models discovered differ. For example, although we did not walk through the process in detail, a similar, large design space must be navigated.

When combined with modern distributed storage systems, the pricing of today's commodity hardware is resulting in storage systems that can be configured with high core counts, plenty of RAM, and fast local devices. However, today's storage interfaces do not expose interfaces that allow these resources to be utilized directly. In this section we showed how a database management system could make use of programmable storage interfaces to offload query operator execution (e.g. selection and projection) to achieve performance gains compared to a traditional architecture in which all data processing happens locally to the database node.

4.5.5 Declarative approaches

The prototype system we built, Skyhook, is designed to take advantage of a database-specific storage interface built into Ceph. However, the implementations that we experiment with are designed assuming a particular storage system configuration and performance profile. As we presented in the previous section, an implementation that codifies an approach to indexing (e.g. the bulk index update problem we observed) can perform sub-optimally simply by changing the way

that data is partitioned. These indirect effects are highly dependent on system and workload configuration, implementation of interfaces, and design choices of internal storage sub-systems. For these reasons, any static approach to a data interface will need to be continually updated as the system and assumptions change.

Furthermore, the exact decision about running computations in the storage cluster versus moving data back to the database system is a specific case of a more general challenge, and has been covered in related work. While these cost-based decisions may be relatively simple, building the cost models and a system for constructing an optimized execution plan has been studied extensively in the database literature.

The high-level approach to building the interfaces described in this section using declarative specifications is similar to the cases we have seen before with the distributed shared-log. A specification should abstract across issues such as data layout and high-level data distribution and partitioning. However, the set dimensions to such a cost model may be quite different. For example, a data interface that may have a high cycles-to-data ratio may result in run-time decisions that affect where computation is performed depending on the capabilities and current state of the storage system. Finally, a programmable storage system that is capable of hosting a variety of interface types may require a unified approach to generating interfaces. For example, a decision about running a database scan locally versus remotely may be dependent on the run-time state of the system with respect to resource consumed by entirely different set of interfaces. Therefore, it will not be sufficient for these types of decisions to be made in isolation by an application, rather a high-level approach will be required that can reason across application interfaces.

4.6 Structured data management

The scaling requirements of high-performance scientific computing applications have been a driving force in system innovation. With both structured and unstructured datasets that push the limits of scale, innovative data management techniques are always under development to deliver performance from HPC systems. In this section we look at one specific topic in HPC scalability which is checkpoint-restart, and how storage system programmability may provide advantages.

4.6.1 Checkpoint-restart

Large-scale, long-running computations rely on parallel file systems to save periodic checkpoints of application state. A checkpoint represents a globally consistent view of a computation and can be used to restart an application from a known point, following a failure. Since checkpoints require consistency across potentially millions of threads, computations are generally suspended during the checkpoint process. Thus, decreasing the time required to complete a checkpoint can result in immediate increases in compute efficiency. However, a common I/O workload in which all processes write a single file (N-1) is notoriously difficult to optimize due to intra-file serialization.

The Parallel Log-structured File System (PLFS) was developed as a middleware layer to address throughput scalability for N-1 workloads by transparently decoupling writers, and transforming N-1 workloads into N-N workloads in which each process writes a dedicated file. In PLFS, each process writes to a dedicated log-structured file, and avoids the need for finding specific "magic number" tuning parameters [22]. Since the logical view of the file being written—a single byte-stream—is no longer explicitly maintained, each write must be recorded in an

index, and this index must be globally available to all processes when the file is opened for reading in order to identify the log shard containing a particular byte.

The PLFS middleware is implemented as either a FUSE-based file system, or as an ADIO plugin to MPI-IO, and is designed to sit directly above a standard POSIX file system interface. Index and log-structured files are maintained by PLFS, stored in a special container directory that represents the high-level logical file, and all index creation and compression is performed above the file system interface. The scalability of PLFS can be reduced by two primary factors. First, the size of an index can become very large resulting in increased I/O when writing, and memory pressure from the index required by each client when reading. And second, the number of individual log and index files can grow to put pressure on the underlying file system's metadata services [83].

Programmability approach

In much the same way that we presented how the CORFU log abstraction and its optimizations could be replicated in a programmable system, here we show that a technique for scaling checkpoint-restart workloads is also possible to replicate.

A programmability-based approach to implementing PLFS makes three improvements over the middleware-based architecture. First, index creation and maintenance can be handled transparently by the underlying object storage system using a PLFS-specific data interface. Second, metadata load is reduced by avoiding the creation of multiple files per process—this is a result of using the underlying object store for naming, rather than solely relying on the file system metadata management services, and this will be discussed further in Chapter 5. And third, index consolidation and compression can be performed offline, allowing the computation to resume as soon as possible. An overview of the architecture

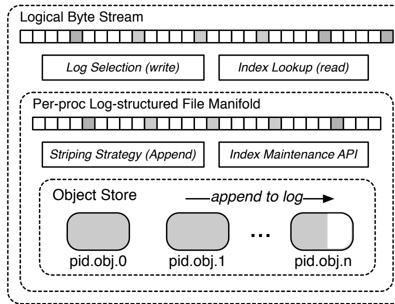


Figure 4.15: The PLFS abstraction hierarchy. The top-level file view maps writes to per-process logs that are automatically indexed by low-level active objects.

is shown in Figure 4.15, where a three level hierarchy is illustrated.

Figure 4.15 illustrates the multi-level design we used to demonstrate a programmability enhanced implementation of PLFS. The outer abstraction, labeled *logical byte stream*, represents a single logical file into which multiple processes are writing checkpoint data, and is identical to the file view presented by PLFS. This is analogous to the high-level log abstraction presented in Section 4.4. The I/O that clients generate against this file is translated by the client transparently onto underlying object storage that uses a custom data interface.

The core abstraction in PLFS is an auto-indexed, log-structured file that transforms writes at logical offsets into efficient object append operations. We constructed a PLFS-specific data interface that is used to implements this abstraction, which performs automatic indexing. In its simplest form, clients stripe data across a set of append-only objects that the application maintains in a dedicated namespace, using a basic naming scheme to preserve the append order (e.g. O_0, O_1, ...). Writes to logical offsets are translated into writes to objects, and once objects reach a certain size threshold a new object is allocated to extend a process' append sequence.

We constructed a PLFS-specific object interface that acts as a building block

for the log-structured file abstraction used by PLFS clients, and is responsible for managing the logical-to-physical mapping using the internal omap indexing facility. Figure 4.16a illustrates the functionality of the data interface when a write is received. A position for the write is determined, and the logical-to-physical mapping is recorded directly in the index. In a similar manner, a logical offset is read by using the index to form a view over the data contained in the object.

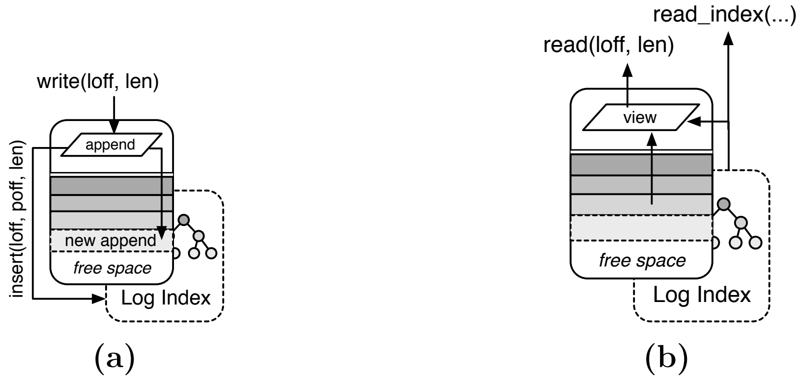


Figure 4.16: Data interfaces used by PLFS file abstraction. An index is automatically generated, and logical views can be constructed.

Immediately following the completion of a checkpoint using this design, opening a file for reading can be expensive: the global index is uncompressed and fragmented across all of the objects composing the entire high-level file. To address this issue the data interface may perform compression on the index to reduce the amount of data returned to clients opening a file.

Index compression by merging. The set of index entries corresponding to the objects of a process log are compressed using two strategies implemented as a pipeline, shown in Figure 4.17. The first strategy is *merging*: A PLFS index fundamentally consists of a set of 3-tuples (logical offset, physical offset, length) that map the logical offset of an extent to its physical location. The current version of PLFS performs basic compression by *merging* adjacent entries that correspond to a contiguous logical extent. For example, the index entries for two 100-byte

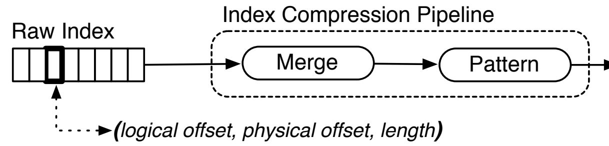


Figure 4.17: Index compression pipeline.

writes at offsets 0 and 100 can be replaced by a single 200-byte index entry at offset 0. This is implemented in PLFS by buffering index file updates, merging when possible, and periodically flushing the buffer. The first stage of the compression pipeline we use performs the same merge-based compression, and achieves the same compression ratio as if PLFS used an infinite buffer. However, in practice the periodic buffer flushing will introduce only a small amount of inefficiency to the resulting compression ratio.

Index compression by pattern recognition. The second type of compression is not performed by the current version of PLFS, and utilizes pattern recognition to identify regularity in the I/O pattern, replacing a series of index entries by a compact representation when a series of entries matches the pattern. The pattern that a series of index entries must match is the same as a standard strided I/O pattern, (logical offset, length, stride, count), plus a starting physical offset. Thus, the pattern can be expanded using the formula, $O_l + i * S, i \in [1, count)$, where O_l is a starting logical offset, S is a constant stride size, and *count* is the number of extents represented by the pattern. Other approaches to pattern recognition and compression have also been explored in the context of PLFS write indices [57].

We implemented the log-structured data interface in Ceph using the techniques described throughout this chapter. Initial micro-benchmarks using a single OSD show between 11% and 17% throughput overheads for an append workload generated by the low-level data interface I/O transformation. However, the workload is

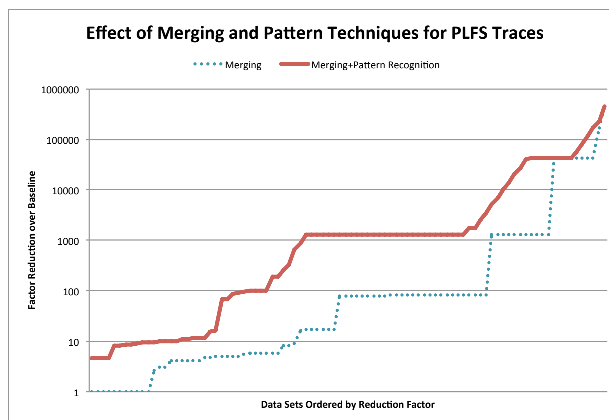


Figure 4.18: Index compression ratios for 82 PLFS traces.

generated by a single client using a closed-loop workload, and thus the source of the overhead is likely to be latency. Increasing the load and optimizing physical layout of the index to reduce the number of additional I/Os is expected to reduce the throughput overhead.

We have performed an analytical evaluation of the reduction factors obtained using index compression techniques mentioned previously. We used PLFS traces from a mix of 8 applications and I/O benchmarks using between 8 and 512 processes [7]. The reduction factors for both techniques are shown for 82 PLFS traces in Figure 4.18. For each trace the effect of merging was reported, and the effect of applying the pattern recognition technique after merging. The two curves show the distribution of reduction factors obtained by each technique.

There are three interesting modes present in the graph. At the high end pattern-based compression offers little to no benefit over merging, due to the I/O patterns that have a high degree of sequentiality per process. At the low-end, strided workloads with small writes are incompressible by merging, but the I/O pattern is discovered and pattern-based compression is applied. In the center the distributions are parallel, indicating workloads with large-scale patterns that can be detected, and per-process small sequential writes that can be merged.

4.6.2 Summary

Scientific data, such as the data that may be managed in files stored in PLFS, is often highly structured. This is an important class of data as it drives increases in the size and scale of high-performance machines used in research. In this section we examined how programmability could be used to emulate one middleware approach to accelerating I/O for scientific applications, and add additional enhancements such as index compression.

Beyond the PLFS use case described in this section, there is a wide range of future work related to the storage and management of structured data and interfaces designed specifically for structured data. For example, special purpose systems such as SciDB use domain-specific optimizations, and expose a declarative interface for interacting with the system [28]. Other frameworks such as Legion and Chapel contain detailed information about the structure of data managed by applications [20, 31], and can make this structural information available to a storage system. Additionally, services not specific to structured data but which support the PLFS use case include scheduling offline work such as building and compressing PLFS index structures that can be completed asynchronously to an application's checkpoint process.

4.7 Durability

Of all the properties that characterize storage systems, durability is perhaps the most prominent and pervasive. The property of durability ensures that data entrusted to a storage system today may be retrieved in the future. In this section we use the term durability in the broadest sense of the word, so as to include properties of storage media like persistence, as well as availability through

redundancy at both the hardware level and the data level through mechanisms such as replication, RAID, and failure recovery.

Durability has a direct and non-negligible impact on the performance of applications and storage systems. For instance, applications that expect writes to be consistent and durable may be forced to wait until data is synchronously written to multiple independent storage locations. Even if the underlying storage media is fast, network and media I/O latencies can accumulate. Other resources like the CPU are also involved in providing durability, especially for mechanisms like erasure coding schemes, or data scrubbing techniques.

Applications manage a broad array of data, and not all data need be subject to the same level of durability. By allowing applications to define the level of durability required for each piece of data, developers can make trade-offs between durability and other metrics such as performance and price. For example, Amazon AWS and Google Cloud offer reduced redundancy data storage that allows developers to trade-off durability (i.e. the likelihood of data loss) for reduced pricing. In this section we will look at how durability can be adjusted in domain-specific ways to offer a trade-off for better performance, and how mechanisms that are used to provide data durability and availability can be repurposed to provide redundant services required by high-availability applications.

The space of possible configurations directly related to durability can be very large, covering many different strategies for replication and erasure-coding that each offer different trade-offs. And the space becomes even larger when considering different consistency models like eventual consistency. Indirectly, higher-level features offered by a system like data snapshots and versioning, and failure-recovery are all intimately intertwined together with the ability of the storage system to offer a particular level durability.

The ability to fully untangle these complex properties to allow them to be programmatically recombined is beyond the scope of this thesis. We focus instead on two basic aspects of durability to demonstrate some of the possibilities, and conclude with a discussion about some additional future directions. We will examine persistence and availability as two system properties that can be used to reproduce application-specific optimizations and common service requirements.

4.7.1 Persistence

The persistence model of storage media is a crucial parameter in the design of a storage system. For example, systems that provide strongly consistent, durable I/O may synchronously replicate data across persistent media such as disks or SSDs whose data will survive power interruptions. However, some systems may temporarily store data on volatile media to offer better performance with reduced guarantees. Even though non-persistent memories like DRAM are frequently used in storage systems as very fast media to cache frequently accessed data, or data that can be easily recovered, storage systems tend to outwardly expose interfaces that provide only persistent and durable storage of data. Applications with more diverse needs are expected to provide their own solutions to data caching and handling of non-persistent state by either implementing a custom solution or using an existing, specialized storage system such as memcached [47].

Next we will consider the use of programmability to build an equivalent service to that of the sequencer used in the CORFU high-performance log. This service depends on both the performance of volatile memories, as well as an application-specific protocol for recovering the state of a failed sequencer.

The CORFU sequencer

CORFU utilizes a service called a *sequencer* that is used to provide a high-throughput assignment of log positions to clients that are appending to the log. The sequencer service is able to achieve high-performance because its state is stored entirely in volatile memory, allowing a lightweight implementation that is accurately described as a network-attached counter. Implementations of a CORFU sequencer using UDP can achieve 100K-1M sequence assignments per second.

In this chapter we have shown how data interface programmability can be used to support many different aspects of domain-specific storage services. Data interface programmability can also be used to define object interfaces that behave like basic RPC services, exactly the type of interface exposed by the CORFU sequencer. Fundamentally a CORFU sequencer manages a single monotonically increasing counter representing the current tail of the log, and exposes an interface for retrieving the current value of this counter, and an interface for incrementing the counter. By storing the counter state in a Ceph object, data interface programmability can be used to build the equivalent of a CORFU sequencer service by relying on the atomicity of data interfaces. As we have seen in previous sections, one of the first tasks in building such a service is to determine the physical design and how state will be managed. The existing internal interfaces for managing state in Ceph all force data to be stored durably, such as replicating the state across nodes in the system.

This limitation is a product of the assumptions made by the storage system about both the way state is managed, and the common guarantees that client applications expect. Building a sequencer service interface by storing the state of the sequencer using durable interfaces will inherently limit the performance of

the sequencer to the performance of durably storing data (e.g. replicating the state across multiple nodes). In order to achieve durable I/O performance on the order of 100K-1M operations per second using replication across nodes, extremely performant networks and media would be required. Thus we need both volatile memories like DRAM, as well as a way to manage state without the normal overheads of mechanisms such as replication.

By foregoing the normal persistence property of media to store state, higher performance can be achieved. Consider the initial throughput of the object-based sequencer shown in Figure 4.19. This sequencer is implemented as a custom data interface in Ceph, but the sequencer does not store state on the underlying persistent media such as a disk—it only uses an in-memory cache for state management. Compared to a target throughput of 100K-1M sequences per second, the performance falls far short. However, there are two aspects to consider. The first is that this performance is a 2x improvement over storing the data using a persistent interface such as omap. Eliminating the latencies associated even with a fast SSD as in this case provides a performance improvement, but the underlying server component of a Ceph OSD, and the underlying network should inherently be capable of at least an order of magnitude more throughput.

The key to understanding the poor throughput is that systems like Ceph have been built in the long shadow cast by slow media like spinning disks, and even SSDs (compared to the performance of today's networks and CPUs). This has resulted in systems in which the cost of code path length and context switches have been dwarfed by the latencies of media. For example, a generic operation handled by a Ceph OSD must first be picked up by an available worker thread, and is then subjected to a laundry list of checks and balances. The code path related to these checks include such things as verifying the state of an operation

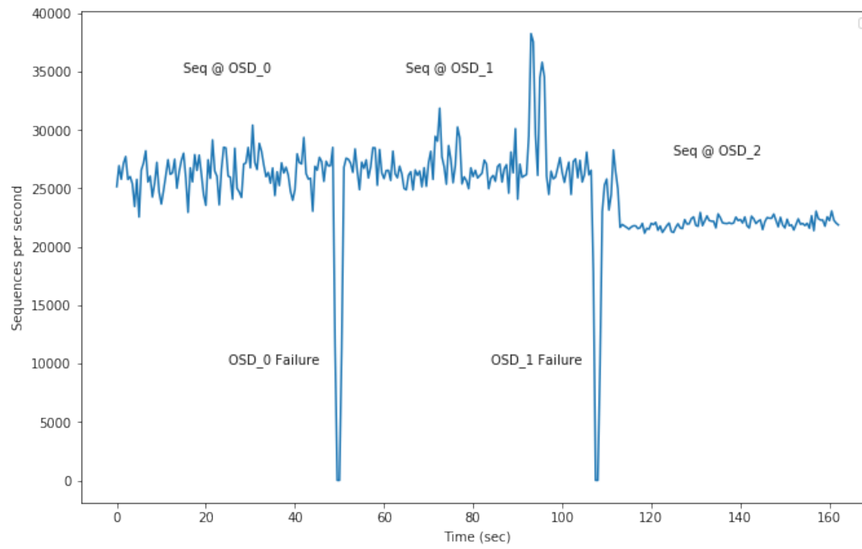


Figure 4.19: A sequencer service implemented as a data interface. Sequencer service failover is achieved by using the availability features found in the storage system. Above we have configured the service with 2 replicas such that after 2 failures the service is still functional.

against the state of the OSD along dimensions like consistency requirements, the state of on-going data recoveries and scrubbing, I/O priorities, permissions and security, as well as data management features like snapshots, cloning, and tiering. All of these checks culminate with the construction and life cycle management of a transaction context that an operation executes within.

Crucially, every client operation is handled by Ceph in this way, even the operation for our data interface implementing the CORFU sequencer protocol, a protocol whose state is a single integer that can be stored in volatile memory, and which in principle could avoid much of the complexity that other operations must handle. When this knowledge is combined with the understanding that Ceph enforces a serialization of operations on individual objects, we can begin to see where the source of reduced throughput for our sequencer interface lives.

Improving this situation is a matter of basic engineering and architecture changes internal to Ceph. For example, providing a fast path for volatile state

management can be added that allows the complexity associated with the work of managing data assumed to be durable to be circumvented. However, consider this design in the context of the CORFU sequencer. Such a service is inherently a *named* endpoint whose name should survive system failures. If state is managed in volatile RAM, then there is a challenge related to how Ceph will manage failure and recovery of the service. In the next section we examine availability in this context, and show how it can be customized to provide fail-over functionality for services such as a CORFU sequencer.

4.7.2 Availability

The availability of a storage system refers to its ability to continuously operate even when failures occur. Availability is achieved through both data and hardware redundancy. For instance, a common replication scheme is to make N copies of data on N different physical nodes, allowing up to N-1 failures while still providing access to data. Various configuration parameters effect the exact scenarios in which data is available and updates are allowed, for instance a system may require that when failures occur that at least two or more nodes are active to continue to operate.

Durability and availability are extensively researched topics in the storage system literature. In this section we focus on one basic, but common form of availability: N-way replication. This scheme, an option for durability in Ceph, provides the equivalent of a primary-backup system using N-1 replicas in which a failure of the primary results in a replica being elected as the new primary. In this mode, a full copy of an object is maintained and synchronously updated on all replicas.

In the previous section we discussed how an object storing a counter can be

programmatically configured to behave as a CORFU sequencer. If we consider an object in this way to represent a service, then availability mechanisms provide the equivalent of service recovery, a task often implemented using an external system such as a quorum of Zookeeper nodes. In this section we discuss using programmability to achieve CORFU sequencer failover, which is complicated by the fact that object state is maintained in volatile memory for application-specific optimizations.

Sequencer availability

There are three challenges in designing a CORFU sequencer using the data interface programmability offered by Ceph. First, as we have just discussed, an important optimization technique is achieved by storing the sequencer state in volatile memory. The non-persistence of the sequencer state also enables optimizations such as eliminating redundancy mechanisms like replication. The remaining challenges are related to the sequencer as a service, namely maintaining the service availability while retaining this key optimization technique.

The second challenge is related to the persistence of the service endpoint name. An application configured to use a particular sequencer service should be able to continue to access the service using a persistent name, despite the state of the sequencer service being stored in volatile memory. And the final challenge is the initialization of the CORFU sequencer state. Following a failure of a node acting as the primary for an object configured as a CORFU sequencer, all in-memory state will be lost, including the state of the sequencer. However, since this state is not stored redundantly, any replica that is elected as the new primary must take steps to initialize the sequencer state. We will now look at these two aspects.

Sequencer service naming

Objects in a system like Ceph contain both application-level data, as well as internal metadata that tracks numerous pieces of state such as the object name, and statistics. In order to provide transparent fail over to clients, an object in Ceph may be accessed with the same name regardless of the node in the system acting as the primary. Ceph provides transparent fail over and retries on behalf of a client.

In order to provide service failover for the CORFU sequencer, clients should be able address the service by the same name after a node has failed and a replica has been elected. This implies that even though the application-level state of the sequencer (i.e. the counter) is not replicated, metadata about the object, including its name, must be replicated in order to provide durability and availability of the service endpoint itself. The overhead of tasks such as creating a new object context and replicating this context including metadata like object names is borne when an object is first created. Following creation of the object context, application-level volatile state can be managed without additional replication.

Sequencer recovery

Prior to responding to client requests, the in-memory state of a CORFU sequencer must be initialized. For a new log, this state can be a constant (e.g. 0 or 1). Initializing the state of a sequencer for a log that has already been created and been in use requires an application-specific protocol that examines the objects that store log data. Recall from Section 4.4.1 the CORFU data interface defines several methods such as *read* and *write*, as well as methods such as *seal* which is used to determine the maximum log position written. A sequencer state is recovered by computing the maximum log position written across all possible objects

that may store the tail log entry, and initializing the state to be the computed maximum. In practice computing the maximum position requires contacting a subset of the objects (e.g. the objects contained in the latest epoch), and they may contacted in parallel allowing fast recovery.

Figure 4.19 shows what this looks like in practice. A CORFU sequencer is constructed as a domain-specific object interface and replicated across three Ceph OSDs. In the experiment shown a client continuously requests the next log position from the sequencer service. The graph is annotated to show the following scenario. Initially the sequencer state is managed by OSD 0, and approximately 50 seconds into the experiment we forcefully shutdown the primary OSD. Within a few seconds the OSD 1 replica has taken over as the new primary and service resumes for the client. This process is repeated once more 110 seconds into the experiment at which point OSD 2 takes over after OSD 1 fails. In these experiments we do not fully recover the sequencer state. This process is application-specific and can be integrated using a callback mechanism into these recovery routines, and the CORFU recovery protocol can be run in parallel to object fail over. In our experiments we use a fixed sequencer state during recovery.

The amount of time required to recover is dependent on many factors. For example, if a failed OSD manages a large volume of object data, then the object representing the sequencer service may be treated by the storage system identically to other objects storing data and recovery may be delayed depending on the order in which object access is restored. This can be tuned by applications using the Ceph pool abstraction to map the sequencer object onto dedicated hardware creating fail-over configurations where the only objects to recover are sequencers or other objects with a small amount of state. Another aspect of recovery performance are timeouts associated with marking OSDs as failed. Ceph includes many

tunables that can be configured to control how failover behaves.

4.7.3 Summary

Deconstructing services provided by a storage system, such as those providing storage durability, into its constitute components can reveal general purpose building blocks that support application-specific data management needs and optimization strategies. In this section we saw how persistence can be applied to data managed by a system to provide performance enhancements. We also showed that when data interfaces are thought of as services, that existing availability features in storage systems can be used to provide the common requirement of service fail over and recovery using domain-specific recovery rules. By factoring out the application-specific aspects of service recovery, the existing availability mechanisms enable applications to create recoverable services and focus only the application logic.

Even low-level mechanisms such as replication can be used to achieve domain-specific optimizations. For example, recent work has shown that replicas can be used to store scientific data, or relational database indexes, each with different physical designs optimized for a variety of workloads [36, 62]. Understanding how these optimizations can be integrated into the storage system as application-specific customizations on existing sub-systems will allow the scope of programmability to fully expand into aspects of the system like durability.

4.7.4 Declarative approaches

Recall from Section 2.4 the declarative specification of the CORFU data interface. Notably absent from that specification was the sequencer component. Interestingly, the CORFU recovery protocol for sequencer state is also capable

of being expressed declaratively. The sequencer state is computed from only the durable state found in objects storing log entries. Therefore the state of the sequencer can be thought of as a cached view (i.e. a function) over the durable log state. This allows a declarative approach to utilize the full breadth of existing research on view maintenance in database management systems.

4.8 Graph processing

Throughout this chapter we have used the CORFU shared-log abstraction and other services as motivating examples for the development of programmable data interfaces. In this section we are going to discuss how programmability can be applied to problems related to graph processing. Our driving example is a log-structured database that can benefit from a data interface that understands the structure of fine-grained metadata stored in the log. And this example is unique beyond its data model. We will discuss how it is important for data models and interfaces built with programmability to be composable. In particular, we describe a database storage engine which stores all the entire database through the CORFU shared-log abstraction. As we have seen, the maintenance of a relatively basic abstraction such as a log can be difficult to build and maintain. By composing interfaces these complexities effectively have a multiplicate affect on the size of the design space. For this reason, this section also serves to further progress the argument for declarative programming techniques

The shared-log abstraction has been shown to be a powerful building block for constructing cloud-scale data management systems [17, 116, 117]. One important technique found in log-structured systems is the use of metadata to record relationships between elements in the log (e.g. back pointers [92]). However, existing high-performance log implementations expose storage interfaces that use

opaque data types and passive interfaces, forcing applications to manage and interact with metadata—such as following a pointer—at a coarse granularity. For example, systems like CORFU provide a scalable log interface but operate at the granularity of a flash page [18]. This can lead to I/O amplification, and increased latency for many applications.

One compelling instance of a data management system built on a shared-log such as CORFU is the Hyder distributed database system [25]. Hyder is a transactional key-value data store that is structured as a copy-on-write tree, and stored entirely in a shared-log. In addition to application data, Hyder stores within the log all of the fine-grained structural information such as tree nodes and pointers, as well as metadata that is maintained for transaction processing [23, 24]. In Hyder, database nodes function as a partial cache of the tree stored in the log, and handle cache misses by using metadata to locate and read sub-trees from the log storage.

Despite Hyder maintaining fine-grained structural metadata, the CORFU storage interface restricts access of log entries to the granularity of a flash page. In our testing we found that for non-cached point queries this restriction resulted in increased read amplification, which can be significant when reified, depending on the size and distribution of key-value pairs in the database. As non-volatile memories become more widely deployed, their byte-addressability will exacerbate these inefficiencies unless interfaces are designed for fine-grained, intelligent data access. However, log abstractions that provide fine-grained access to application-level data can also result in inefficiencies due to the amount of I/Os required to traverse data structures and data relationships encoded into the stored data. This section introduces *zqlog*, a log abstraction that allows applications that manage log-structured data to navigate these types of trade-offs.

To start, consider the process of executing a point query in Hyder which reads a single key from the database. Since Hyder is structured as a tree, a path exists from the root (R) to a node containing the target key (K) that must be traversed. However, the physical organization of the nodes along this path is dependent on the workload that created the database, and in general these nodes may all be stored in separate log entries. In order to traverse the fixed path from R to K, the log entries containing nodes along this path must be read. The read amplification that we observed was a result of reading log entries that contained additional data irrelevant to the point query (i.e. other sub-trees). One solution is to create a log abstraction that allows direct access to individual application-level data elements, such as single nodes in the tree managed by Hyder. However, we found that for point queries in Hyder, optimistically caching all nodes in a log entry was an important optimization for reducing the frequency of log read operations. This example highlights a fundamental conflict between the need to publish timely updates to the log, and predicting the I/O pattern of future workloads.

The zqlog interface that we introduce in this section is a shared-log abstraction that does not force applications to define a fixed physical layout and access plan for each log append. The interface seeks to allow applications to apply a schema to each log, and combined with the flexibility afforded by programmable storage techniques, application-specific queries can be run in the storage system allowing features such as server-side pointer chasing and intelligent prefetching. We do not fully develop a specification for applying and using a schema. At the end of this section we discuss how a schema is important in this work, but leave that to future work related to declaratively building interfaces. The flexibility provided by a query interface to a shared-log allows applications to navigate the trade-off between fine-grained and coarse-grained access for common log-structured data

management scenarios.

In the remainder of this section we will discuss the problem space of Hyder running on a shared-log, and how a new interface to the shared-log can benefit the Hyder application. While we do not develop in this section a fully automated solution, we do derive a simple cost model that demonstrates the need for an automated approach to building data interfaces declaratively for this use case.

Background. We assume a system exposing a CORFU-like distributed shared-log abstraction, which has been covered in detail throughout this chapter. One aspect of the original CORFU system that is salient to the discussion in this section is that CORFU maps log entries to coarse-grained flash pages of storage devices. We will argue in this section that this leads to inefficiencies when managing fine-grained metadata.

Applications make use of the shared-log abstraction in many different ways that become apparent when examining how applications handle data management tasks such as encoding and storing data relationships (e.g. pointers) in the log. For example, Tango is a system for building cloud-based metadata services that exposes a multi-stream abstraction on top of a single shared-log [17]. This abstraction is used to logically partition updates to Tango services while depending on the serializability of the underlying log to implement consistent updates across services. Tango implements this abstraction by using back pointers that thread together log entries belonging to the same stream. These pointers are collected into groups, and *stored alongside application data*, as low-level metadata in each log entry that is loaded and used during start-up and recovery. Since existing log implementations provide I/O at the granularity of a log entry, Tango is forced to load application data—as opposed to only metadata—while reconstructing stream membership. Next we will examine this issue in a more complex application.

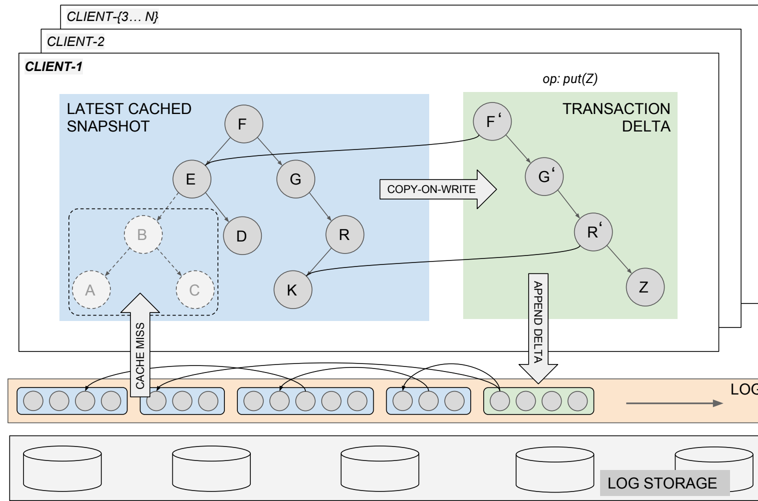


Figure 4.20: High-level architecture of Hyder in which clients append deltas to a shared-log, encoding the effects of a transaction. Uncached nodes are paged-in from log storage.

4.8.1 Motivation: log-structured database

Hyder is a distributed, transactional key-value database stored entirely in a single shared-log [25, 24, 23]. The system is designed to run on a log that exposes an opaque I/O interface, but Hyder manages complex data relationships. As we will see in this section, this can lead to various inefficiencies that motivate the design of the *zqlog* interface.

As shown in Figure 4.20, the Hyder database is structured as a copy-on-write tree, and stored in the log as a sequence of *deltas*. Deltas are generated by independent clients executing transactions against their latest cached snapshot of the database. Each delta records the changes made by a transaction, and contains a mixture of metadata for structure sharing (i.e. sub-tree pointers), as well as annotations on tree nodes used to implement efficient transaction concurrency control. Clients process the log in order, examining each transaction delta for conflicts. When a transaction is found to have no conflicts it logically commits, forming a new database snapshot. Critically, log processing and conflict analysis

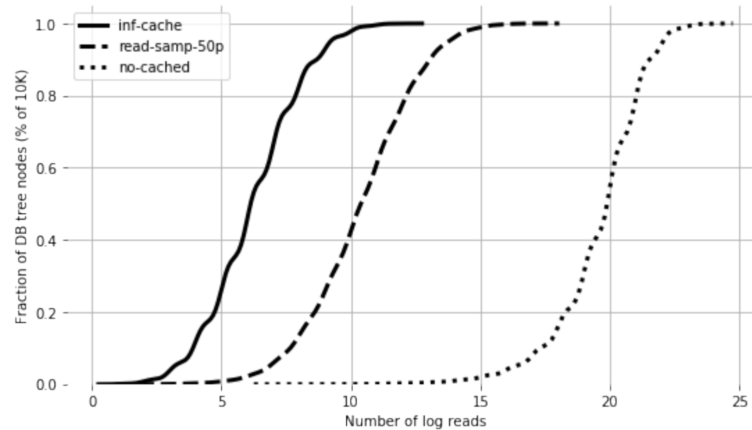


Figure 4.21: The number of log entries read to access any node in a test database consisting of 10K entries, under a variety of scenarios.

are deterministic, so every client makes the same set of commit and abort decisions independently, and can do so in parallel.

Using a data-sharing architecture, Hyder clients can be viewed as managing partial caches of the entire database stored in the log. Shown in Figure 4.20, the tree nodes A, B, C are greyed out indicating their presence in the database, but absence from *client-1*'s locally cached snapshot. During transaction processing, sub-trees that are missing from a client's cached snapshot are paged into memory on-demand by reading directly from the log. Missing sub-trees are read from the log using persistent pointers embedded into each tree node that record the location of the node's children in the log.

Consider the process of accessing a single key-value entry in Hyder. Starting from the root of a snapshot, a path exists to the target key that must be examined. And when a node along the path is not resident in memory, it must be read from the log in order for the traversal to make progress. For instance in Figure 4.20, locating node C requires first reading node B into memory from the log.

We explore the costs of this system design by reporting I/O metrics for point queries executing against a test database constructed by inserting 10K uniformly

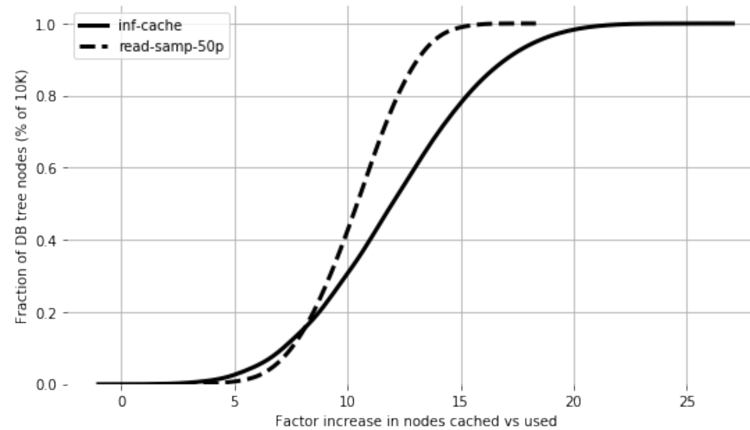


Figure 4.22: The factor increase in total nodes read over nodes required to satisfy a point query for any key in a database.

distributed random keys. We run all experiments in a database system called CruzDB [2] that is modeled after Hyder. One important difference between the two systems is that CruzDB stores transaction deltas that include the effects of rebalancing, resulting in log entries that generally contain more tree nodes. The experiment we ran measured the number of entries read from the log in order to fetch a given entry. We repeated this experiment for every key in the database, before which the client cache was cleared. Figure 4.21 shows the cumulative distribution of log reads required to access any key under various scenarios. For example, the *no-cache* distribution gives the worst-case number of log entries read, which occurs when no tree nodes along a path are cached. Intuitively, this is the distribution of node depths in the database, which is structured as a balanced binary tree.

In comparison, the *inf-cache* distribution in Figure 4.21 shows the cost to access any key in the database provided that when a log entry containing a delta is read to satisfy a cache miss, all nodes in that delta are optimistically cached. This reduces the number of log reads because a delta often contains more than one node along the path to a target key. But this reduction comes with the overhead cost

of reading entries that may not benefit the current query. Figure 4.22 quantifies this overhead in CruzDB by plotting the number of nodes optimistically cached, but never used in support of a given query. In the *inf-cache* scenario roughly half of all keys that are accessed cause a logical read amplification of 10x or more.

As we will see in the next section, even fine-grained passive I/O interfaces offer little in the way of a solution to this dilemma. And so far we have only considered the simple, constrained case of single key point queries; this issue exists in other data management operations such as log scans, segment cleaning, and data expungement. One approach to navigating the trade-offs between fine-grained and coarse-grained log interfaces is to construct domain-specific log interfaces using techniques from *programmable storage*, which we describe next.

4.8.2 Programmable storage approach

To demonstrate how programmable storage can benefit our motivating example of a log-structured database system we created three application-specific log access methods, listed in Table 4.1, along with the standard *read* log interface. When the *read* interface is used by the database system, all of the data at a log position is unconditionally returned. In this case the database optimistically caches the contents, resulting in *inf-cache* case we observed in Figure 4.21. Now consider the application-specific *read-node* interface that accepts an index parameter specifying a specific node stored at the log position. This interface supports fine-grained access to tree nodes, but only returns the exact node requested. Using this access method for tree traversal corresponds to the *no-cache* case in Figure 4.21 since no additional nodes contained in a transaction delta are returned to a client and made available to be optimistically cached.

The remaining two application-specific interfaces listed in Table 4.1 provide

read(pos)	read all data stored at a log position (**standard log interface**)
read-node(pos, idx)	read a tree node from a log position
read-samp(pos, idx, pct)	read a tree node and return *pct*% random sample of other nodes
read-path(pos, idx, key)	read a tree node and return local nodes along path to *key*

Table 4.1: Application-specific log access methods, and a standard log *read* interface.

more flexibility. The *read-samp* interface reads a target node plus a random sample of co-located tree nodes. The impact of using this interface can be seen in Figure 4.21, where the distribution labeled *read-samp-50p* shows the number of log reads required to access any key in the database when 50% of nodes in a log entry delta are returned and optimistically cached. Adjusting this sampling rate has an affect on both the cache hit ratio for point queries, as well as read amplification, as can be seen by the *read-samp-50p* distribution shown in Figure 4.22. The final interface that we tested—*read-path*—returns a target node like *read-node*, and also accepts a key which is used to guide the selection of co-located tree nodes such that only those along the path to the target key are considered. Creating semantically rich interfaces like this can be powerful; this interface achieves the same cost as *inf-cache* in Figure 4.21, while eliminating read amplification for the given query.

The three application-specific interfaces that we have presented are far from a covering set that would benefit our database example, let alone a broad set of log-structured applications. For example, the semantics of *read-samp* are that a uniform sampling of nodes is returned, but this is non-optimal since a given path to a target key will not intersect with two sides of a tree. Instead, a better semantic for point queries would be to return a sampling that favors nodes closer

to the root of a tree delta. In addition to other data management scenarios such as garbage collection and data expungement, low-level optimizations can be made by exploiting storage locality, allowing dependencies and relationships that *span* co-located log entries to be examined, such as traversing inter-delta tree paths.

4.8.3 Summary

Like previous sections in this chapter, we have shown the benefits of using domain-specific storage interfaces. In this case we demonstrated how a domain-specific interface to a log-structured database could be used to perform remote traversals of a database stored as a copy-on-write tree structure. We showed that an interface can be developed to control prefetching and reduce read amplification by using database query parameters to guide the traversal of the tree structure, and we showed a cost model that demonstrates the trade-offs that can be made.

4.8.4 Declarative approaches

There are several aspects to the log-structured database and corresponding data interfaces described in this section that are relevant to declarative approaches to interface development. Perhaps the most interesting aspect is the need to effectively be able to compose domain-specific interfaces. For example in this section we showed how a log-structured data interface that exposed application-level data structures and access methods could be used to enhance performance. However, the data interfaces specific to the example used in this section were *added* to the existing CORFU log abstraction. An ad hoc approach to programmability would thus likely require an approach that duplicated the entire CORFU customizations, or relied on difficult to maintain interfaces built by punching a hole through many layers of system software. Instead, we believe a promising direction is utilize lan-

guages to assist with these challenges. For example, by allowing applications to apply a schema to a CORFU log, new interfaces could be defined by queries that replicated the original interfaces as well as interfaces whose queries provide the semantic information required to derive optimizations such as push-down processing and remote pointer chasing. And as we saw in Section 2.4 additional interfaces on top of the CORFU abstraction defined in Bloom can be added using a few snippets of code and optimized together with the entire implementation.

4.9 Conclusion

Previously in Chapter 2 we introduced programmability and argued for the adoption of declarative approaches to building storage services and interfaces. In doing so we relied upon only a small set of examples that illustrated basic concepts and challenges. In this chapter we looked at the applicability of programmability across a wide variety of application domains. We found that storage systems could be used to build a variety of basic services that reduce duplication of functionality, improve performance, and simplify construction of data services.

We examined the use of data interfaces for managing transactional data, and exposing internal storage system resources like CPU, memory, and data management facilities to improve the performance of a database management system. We saw how by controlling the durability associated with an interface and reusing basic availability features we could build a high-availability service with domain-specific optimizations. And finally, we demonstrated how data interfaces could be beneficial to HPC applications using structured data as well as data models such as graphs that can benefit from remote traversals of data structures.

In a majority of the use cases discussed we encountered a very large design space and quickly discovered cost models that influenced the physical design or

runtime decisions. This finding further supports the need for a declarative approach to building interfaces that removes the burden of on-going maintenance by application developers that would otherwise benefit from domain-specific interfaces. In each appropriate situation we argued how a declarative approach to specifying an interface may be derived, but building a full solution that uses declarative interfaces is left as future work.

Future work

The development of the examples highlighted in this chapter has produced a tremendous amount of momentum and breath related to future work. First, we have only touched on a subset of the resources found in a storage system that would be beneficial to expose as points of customization for applications.

For example, in Section 4.5 we described how relational database tables could be decomposed into thousands of small objects and queried in parallel. This can be further improved by reducing the number of network round-trips by implementing data interfaces at a higher abstraction layer that allows objects to be queried as a group. This feature makes use of an abstraction called a *placement group* in Ceph, which also serves as a mechanism for controlling data co-location that is useful in many computations such as dealing with ghost data in HPC analysis [64]. Not all object-based storage systems will necessarily expose such an abstraction, and taking this into account is important. Other opportunities include scheduling offline work like data indexing (see Section 4.5 and Section 4.6), and allowing data interfaces to handle batches of requests by controlling the queueing dispatch semantics. In Section 4.7 we discussed how availability mechanisms such as replication and failover can be used in domain-specific ways to create a high-availability service. Replication itself is a powerful technique that can be used to create divergent

copies of data each optimized for specific access patterns [62, 36].

These changes represent potentially significant disruption to the architecture and use cases of storage systems that already suffer from multi-tenancy issues such as performance isolation. We would be remiss to not mention that this may complicate even further the challenges of providing features such as quality-of-service guarantees. However, these challenges have themselves often been explored in the context of storage systems offering a relatively narrow set of interfaces and as such may expose new and interesting challenges.

Chapter 5

Metadata management

An overwhelming majority of storage systems today are built assuming a bytestream I/O interface. This has had a profound impact on software architecture: the inability of applications to explicitly represent domain-specific data models throughout the storage hierarchy has led to the development of middleware libraries that provide data model abstractions (e.g., HDF5, NetCDF), and I/O stack extensions that help circumvent scalability bottlenecks (e.g. MPI-IO, PLFS, IOFSL). While it is common for applications and middleware to influence the design of each other, any co-design process generally ends at the level of file interfaces leaving applications to create their own mappings between application-level data models and a fixed file abstraction. As a result, common tasks such as metadata management that do not fit into the standard model provided by file systems are forced to be handled by applications themselves or delegated to external services.

Throughout this thesis we have examined how components of a storage system can be exposed in a safe way to be repurposed in order to construct domain-specific interfaces. We have shown how this can be used to avoid duplication of services by applications to reduce costs, simplify designs, and increase reliability by using

code-hardened subsystems already available. In Chapter 4 we examined this topic in the context of storage system data interfaces, and in this chapter we do the same for metadata management infrastructure common to many distributed storage systems. Metadata management is completely pervasive in storage systems, serving to describe a system and the data it stores at both a high-level and a low-level. For example, the metadata associated with a POSIX file may describe how that file is distributed within a cluster and its name relative to other names of resources in the system. Similarly, metadata exists to describe high-level components of a system such as the physical nodes that compose a cluster itself, all the way down to the management of data at rest on physical media. We use the term metadata management in this section broadly to encompass both system-level metadata, as well as distributed metadata services for file systems, and the metadata requirements of applications. By exposing metadata services programmatically, an important tool is discovered for construction of domain-specific interfaces using programmability techniques.

Virtually all applications rely on some form of external metadata management. The predominant abstraction for metadata management in large-scale storage systems is the POSIX file system abstraction which provides interfaces to a number of important services for building applications. In its simplest form an application may use the file namespace to attach a name to a basic file resource that it produces or consumes. Some implementations of the POSIX file abstraction provide highly scalable naming for high-performance applications that manage millions of files which can be used to track a large number of application objects. The POSIX interface also defines semantics for how file resources are to be shared among multiple processes simultaneously, and implementations of file systems thus contain important mechanisms for managing shared resources that are used indirectly via

the POSIX interface. Distributed storage systems also depend on the management of metadata that describes high-level aspects of the system such as the set of nodes in the cluster. In this chapter we will show that the metadata management services found in distributed storage system (e.g. distributed file systems and cluster-level metadata management) serve as useful abstractions for building application-specific data interfaces.

5.1 Overview

In this chapter we demonstrate how metadata management in a storage can be used to build domain-specific interfaces by examining two common services. First we introduce *file types* that are extensible interfaces on top of the file abstraction. This allows developers to associate application-level metadata with a file name, providing an important service that applications tend to implement using custom solutions or external services. We demonstrate how an implementation of the CORFU log abstraction can use the file system metadata service to manage log metadata required to bootstrap a client that accesses the log service. We also highlight two applications for scientific data management whose metadata differs significantly from existing file metadata. And when application-level metadata grows too large it can be stored in a data interface to preserve the scalability of the file system metadata service. We show that new services can also be created by reusing internal subsystems found in the file system by building a CORFU log sequencer using a small amount of metadata associated with a file and existing mechanisms for managing distributed shared state.

Finally we show how the management of cluster-level metadata can be extended to include the management of high-level application metadata. We demonstrate this by reusing an internal consensus engine common in distributed stor-

age systems to store, manage, and distribute specifications for data interfaces described in Chapter 4. Although we use this internal subsystem to support programmability itself, the same mechanism and approach can be used by applications to manage data at the same level of abstraction when needed.

5.2 File type interface

We introduce a file type abstraction that allows applications to customize the type of metadata and interfaces associated with names in the file system hierarchical namespace. No longer are files required to implement a POSIX file abstraction, but rather files become a means by which naming and integration with other services, such as cache coherency, can be managed. The file type interface is both a feature and a performance optimization. It is a feature because it allows developers to add support for different storage types, such as how to read new file formats or what consistency semantics to use for a specific subtree in the hierarchical namespace [97]. It is also a performance optimization because future programmers can add optimizations for processing specific types of files into the inode itself.

To demonstrate the benefits to applications of making the file system metadata service programmable we focus on three aspects. First, we provide a naming service that allows the management of application-level metadata using the full power of a distributed, scalable naming service without forcing applications to map names to POSIX files. For example, some applications may wish to create names representing services or data resources accessible through interfaces other than the standard POSIX file interface methods like `read` and `write`. Second, we outline a method for applications to associate metadata with each name in a way that maintains the scalability of the metadata service. And finally, we

show how application-level metadata can be integrated into existing internal systems for managing cache coherency across clients to build new data services with application-specific optimizations.

5.3 Name management

The scalability of the POSIX hierarchical namespace has been a constant topic of research for decades, primarily driven by high-performance computing applications. The file type interface we introduce allows applications to use this naming service without requiring that each name in the hierarchy represent a full POSIX file. While there are other forms of namespace management for file systems that are non-hierarchical, we focus only on hierarchical systems which are the most common and have have been the subject of extensive scalability research. The techniques we develop are applicable to other forms of namespace management.

Applications can make use of the hierarchical namespace to directly represent many common data management scenarios. For example, in Section 4.5 we saw how the Skyhook project stores relational table data in objects, and uses a domain-specific interface to accelerate query evaluation using internal storage system resources. Skyhook may manage thousands of relational tables, and requires a way to manage this space of named data resources. Using file types Skyhook can represent each table as a file organized in the hierarchy, and inherit existing security features already available to users of POSIX files. However, instead of storing file data, Skyhook associates a small amount of metadata with each file name that allows it bootstrap the naming of objects that it controls directly through the Skyhook data interface. Other organization scenarios of data for Skyhook are also possible such as naming individual table partitions. In the next two sections we will examine further how this metadata is associated with each file name in

Figure 5.1: On the left is an example organization of file metadata in which a Skyhook database names individual tables. On the right log0 represents a CORFU log and log1 represents a CORFU log with sub-streams. Each of these examples demonstrates how the file namespace can be used to organize application-specific metadata.

the namespace.

The directory abstraction is also useful for modeling application data relationships. Figure 5.1 shows a diagram of the file system hierarchy that contains the name `/apps/logging/log0` corresponding to a CORFU log resource. This file name associates metadata that CORFU clients use to access the log service managed by the storage system. When a CORFU client accesses such a file it can extract the domain-specific metadata that is needed to bootstrap itself such as the names of objects in the storage system and configuration parameters such as the number of objects that the log is striped across. The name also is used to represent the sequencer service associated with this log, and we will examine how a CORFU sequencer is implemented using the metadata service in the following sections. By managing CORFU metadata in this way, applications may coordinate their access to a log, and discover the sequencer service using the common file system path interface to the sequencer in the namespace without relying on a separate service. Notice in Figure 5.1 another log is present in the namespace as `/apps/logging/log1/`, and unlike `log0`, this log is represented as a directory. The files contained in the directory (e.g. `stream0`) are *streams* contained in the parent log. The stream abstraction is a method for virtualising the log, and is

described in [17]. This organization is convenient for applications because it directly represents the logical relationship between a log and the stream contained in the log, as well as the physical relationship: all stream data is stored together, and all streams share the same sequencer service as the parent log.

This concept of using application-specific file types has been shown to also work for HPC checkpoint-restart workloads that make extensive use of the file system metadata service, as well as for use cases in high energy physics [98] in which entire namespaces may be represented as a compact pattern where the pattern is stored as metadata and transferred to clients. This allows clients to reduce their impact on metadata load by participating in namespace management with reduced communication requirements.

5.4 Scalability

The scalability of file systems has been the subject of extensive research. This has largely been driven by scientific applications that require scalable namespaces that contain large numbers of files per directory, as well as workloads that consist of a high level of update concurrency to the namespace as well as to single files. The semantics of the POSIX file abstraction make scalability challenging. For example, when multiple processes open the same file for writing, distributed file systems must ensure that processes observe consistent updates which requires communication and coordination between the metadata services and the cached data stored by clients. Because of the frequency with which communication takes place in these scenarios, one approach to scalability is to limit the size of the data objects being moved across the network. However, abstractions such as a file inherently grow in size, suggesting that metadata associated with a file (such as block lists) will also grow. The approach taken in some file systems such as

Ceph is to use a fixed-size inode structure and embed parameters in the inode that allow clients to independently map file extents to storage locations. A small, fixed-size inode also allows Ceph to aggressively load-balance metadata across a cluster of metadata servers by making it efficient to move inodes around within the cluster.

Supporting a file type abstraction thus presents a challenge for programmability because applications may associate domain-specific metadata with a name in the file system namespace. In order to maintain the ability of the system to scale, the amount of data stored with an inode must remain small. In Ceph an inode is approximately 1KB in size, including 400 bytes for a directory entry, and 700 bytes for a directory inode. Luckily, many application are able to represent datasets using a small, fixed amount of metadata. For instance, the CORFU sequencer state is represented by a single 64-bit integer, and the CORFU log itself only requires a naming prefix and the stripe width of the log. When the size of the metadata grows large, applications may manage metadata by using data interfaces and storing the metadata in objects (see Chapter 4. Rather than placing a large amount of metadata in an inode, applications may provide a small amount of metadata that links to the data stored in objects.

The types of metadata used by applications is quite diverse. In the next two sections we will briefly highlight two applications and the types of metadata they manage. In the first example we will show how metadata interfaces that expose locality—common in systems like Hadoop—are insufficient in some cases and require a more general treatment of locality. In the second example, we look at a high-performance computing application that can rely on file type interfaces to manage metadata tracking datasets across storage tiers.

5.4.1 Example: locality

Systems like Hadoop MapReduce depend on an important optimization that reduces network data transfers by scheduling computation near the computation's data dependencies [121]. At a high-level Hadoop is structured as two separate systems: a distributed file system, and a separate system that manages computation. There are two additional components that are used to achieve data locality for computation. First, the distributed file system exposes a unique file interface that allows a storage system client to query for the physical locations that store a file's data. Using this locality information that is exposed on a per-file basis, a scheduling component is used to route computations to physical locations that store dependent data locally.

The data locality interface exposed by Hadoop is specific to the byte stream data model. For example, the interface returns the physical location of data given a byte extent of a file. The interface is typically used by requesting the locality of large extents of the file, and it is up to the computation to interpret the data in the given extent. In fact, even Ceph provides a similar interface. While useful, the fixed bytestream-oriented interface does not work well for some applications.

The SciHadoop project is an effort to extend Hadoop for processing scientific data [29]. The structure of scientific data when stored in files can often pose a challenge for achieving data locality using a bytestream data model. For example, a common scientific data model is a multi-dimensional array. However, when such an array is serialized into a one-dimensional file abstraction a fixed layout must be chosen (e.g. row-major or column-major orderings). The result is that a contiguous region of the file at the logical level may be stored at many physical locations. The locality problem arises because the file system interface cannot map between application-level data models (e.g. arrays) and file extents. Figure 5.2

illustrates this challenge and the solution introduced by SciHadoop. The arrow labeled L represents domain-specific metadata that is used to expand the model of locality to cover applications that operate on multi-dimensional array data.

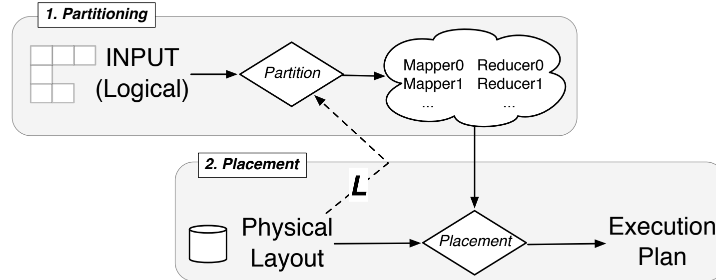


Figure 5.2: MapReduce processes logical partitions in map tasks and matches each map task with physical locations to form an execution plan. The line labeled L is a contribution of SciHadoop which utilizes physical layout knowledge during partitioning

This example highlights how file type interfaces can be used to manage application-specific metadata. The metadata that is used to associate logical views of scientific data with physical extents required to support a locality interface can range from small and compact, to large. When the metadata is large it can be stored in objects directly. In both cases an application-specific file interface is used to expose data locality for an *array* file-type rather than a one-dimensional byte stream. Since the locality mapping is performed internally to the storage system rather than hidden within the application, the array file type interface for locality can be consumed by other applications without duplicating the mapping rules.

5.4.2 Example: scientific data

In this section we highlight a large-scale use case for application-specific metadata management that originates from work scaling high-performance computing (HPC) and scientific data management.

Consider the introduction of additional storage tiers to a typical HPC environment, where each tier may differ in capacity, bandwidth, and latency. Such an expanded storage hierarchy presents many opportunities for sophisticated storage strategies such as hiding latency, intelligently handling low-memory situations, performing I/O staging, and offloading data-intensive compute tasks. Unfortunately it can be difficult for applications to fully exploit the storage hierarchy when data must be managed explicitly by application developers.

For example, an application running on a system with a deep storage hierarchy may integrate knowledge about analysis tasks into its checkpointing strategy by storing one component of application state (e.g., a field associated with a grid point) on a fast tier composed of SSDs for a pending visualization workflow, while the remaining grid fields are placed in a capacity tier for resilience. The data management challenge involved in this example is difficult to solve. In current systems, the application would be required to split the data structure into multiple files, store the data in separate namespaces, and manage consistency and tier migration. While middleware is capable of performing tasks such as complex data mapping and sharding, the data management tasks required to track asynchronous updates to application state across a heterogeneous memory and storage hierarchy while conforming to the consistency requirements of the application are beyond the scope of current I/O libraries that provide I/O optimizations or flexible data serialization strategies. If such complex data management is embedded in the structure of an application, migrating to a new system may require invasive changes to the application that prevent adoption in the first place.

On-going work is examining these data management challenges in the context of next-generation exascale systems that are currently in development by government and industry. For example, by eliminating the POSIX file interface appli-

cations are managing consistency themselves and are able to achieve important optimizations such as latency hiding, advanced parallel I/O, and optimizations that are enabled by handling data with relaxed consistency [113]. However, as explained, this can be difficult for applications and developers to manage. In this section we consider the use of the Legion runtime system—one system being used to tackle these issues [20].

Legion is a data-centric parallel programming system for portable high-performance applications. Importantly, Legion fully controls an application and manages virtually all of the resources of a cluster on behalf of an application. Legion has complete knowledge of the execution and data dependencies within applications that it manages, and can therefore handle complex data management tasks transparently, and optimize operations based on the current state of the system without changes to the application. Legion supports a logical, distributed data model that is decoupled from its implementation on memory or storage and provides the ability to manage the consistency of distributed application state. Legion is unique in that it is a runtime for applications, taking over control of an entire system and managing resources on behalf of an application. However, Legion doesn't manage persistent resources like file systems and externally named resources. In its current form, Legion only manage memory and memory-like storage that does not outlive the execution of an application.

Data Model. Legion introduces and is built upon the concept of logical regions, an abstraction for describing structured distributed data. Logical regions are a cross product of an N-dimensional index space and a number of fields (a field space). Logical regions do not commit to and are distinct from any particular data layout or placement within memories of the machine. A logical region may have one or more physical instances, each of which is assigned to a particular memory

with a specific layout. The data model also supports subdividing the data, either by picking out subsets of the index space or of the fields. We extend this data model to persistent storage by allowing the application data model to directly map to the underlying distributed storage model. This allows applications to specify parts of a dataset that should be persisted or communicated outside the applications.

Memory Hierarchy. Legion models all hardware that can be used to store data as memory (e.g. DRAM and persistent media like disks). The current Legion implementation [111] involves four kinds of memory in which instances of logical regions can be held: distributed GASNet memory accessible by all nodes, system memory on each node, GPU device memory, and zero-copy memory (system memory mapped into both CPU and GPU's address spaces). To support persistence, we have introduced the RADOS memory within the Legion memory hierarchy, which allows Legion to import RADOS objects into its runtime, unifying memory and persistent storage with application semantics [113].

The integration of the storage system with the Legion runtime provides opportunities to make I/O optimization decisions based on both static and dynamic system characteristics. Closer integration of the application level data model and the storage environment allows the runtime to optimize for application specific data decompositions alongside other system characteristics while insulating the application from these optimizations. These optimizations include automatic tiering, sharding, asynchronous I/O, relaxed consistency semantics, and offloading data-intensive tasks to the storage system, like we saw in Section 4.5.

Not strictly related to metadata management, we refer the reader to [113] for a discussion on the benefits to using Legion to implement scientific applications and how Legion can achieve I/O optimizations by handling many aspects of

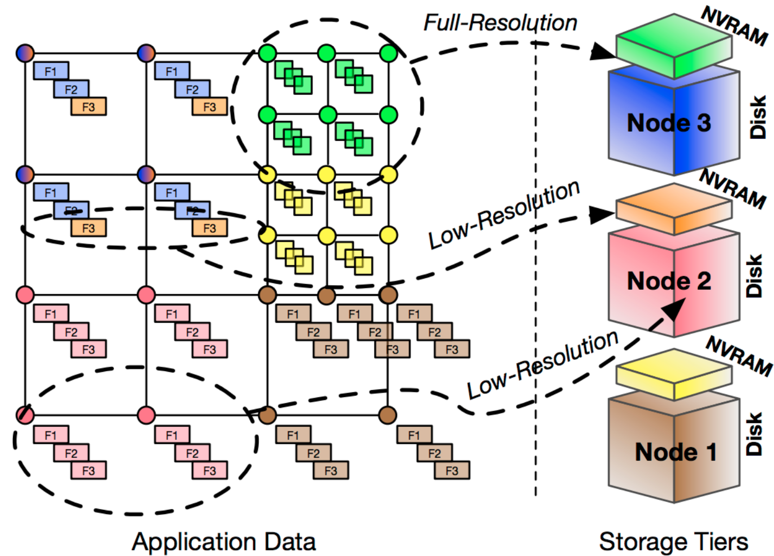


Figure 5.3: Legion data model and mapping of an adaptive mesh refinement application data to distributed and hierarchical storage.

data management itself. The management of applications and application data dependencies when using Legion is similar to normal shared-memory programs: typically a file system is used to track all of the resources. Legion too depends on external services for managing the names of resources. In the remainder of this section we highlight one example application that we have built with Legion that demonstrates the breadth of metadata management that Legion applications rely on, and which the file type abstraction we have proposed can help manage.

Figure 5.3 illustrates our co-design goal by showing how the runtime system with storage integration could effectively map an adaptive mesh refinement (AMR) application data structure into a distributed hierarchical storage system. To the left of the figure is a multi-resolution grid with each vertex associated with three fields (i.e., F1, F2, and F3). To the right of the figure is the integrated, distributed multi-layer storage hierarchy (for simplification, the figure only demonstrates disk and NVRAM tiers on three different nodes).

Regions for AMR applications are often organized at different resolution levels.

In Figure 5.3, we use green and yellow to label regions in full resolution, and use other colors for lower-resolution regions. AMR applications are often more interested in full-resolution regions, which may also come with more frequent accesses. One predominant I/O optimization for AMR application is placing full-resolution regions into a low-latency and high-bandwidth storage layer for better I/O performance. However, the resolution distribution depends on input data, which prevents this optimization from being performed at compile time. Therefore, supporting dynamic data partitioning and data placement at runtime is critical for AMR application performance.

The colors of the fields and storage tiers also represent an application-specific mapping from regions to the persistent storage hierarchy. The coloring in Figure 5.3 demonstrates a possible mapping decision that keeps full-resolution regions (in the top right corner of the grid), as well as field F3 for a subset of lower-resolution regions (marked in orange) in the NVRAM tier, while all other regions are stored on the disk tier. This mapping decision can be completely dynamic and made by the Legion runtime as a result of an optimization strategy, such as staging data on the NVRAM for frequent access.

Legion employs a general method for specifying partitions of logical regions based on colorings that allow for arbitrary data decompositions and layout. A coloring assigns zero or more colors to each element of a logical region. Based on the coloring of each element, Legion provides a primitive partition operation that constructs subregions (partitions) of the elements of each color. As illustrated in Figure 5.3, the multi-resolution grid, which is itself a logical region in the Legion framework, is partitioned into subregions with different colors. The Legion runtime is able to perform different placement, re-partitioning, and layout optimizations for the different subregions.

In Figure 5.3 there are two colorings represented. One coloring is used to facilitate the distribution of the AMR mesh in memory, and a separate coloring on the same data structure represents the desired sharding and tiering decisions that are communicated to the storage system when the data is persisted.

Metadata management

While an extremely powerful system for implementing performance-portable HPC applications, Legion depends on external services for managing persistent data and metadata. Storage system programmability can be used to provide naming and metadata management services from the same storage system that provides bulk data storage for data input and output to Legion applications.

The adaptive mesh refinement examples explored in this section is an example of a complex metadata management challenge. Each level of resolution refinement in the application not only needs to be directly named so that it can be shared with other applications and used to locate artifacts, but will also have a distinct set of metadata associated with it. This metadata includes application-specific information such as grid configuration and coloring information that is specific to the Legion data model, but also includes low-level details such as storage tiers that data is stored on. By utilizing file type facility to manage naming and metadata, the Legion runtime can depend on existing scalable services designed for persistence, and load-balancing in dynamic applications without resorting to external systems.

5.5 Shared resources

In addition to providing a scalable namespace, distributed file systems that implement the POSIX file abstraction often also contain internal services for man-

aging client sessions such as allowing clients to obtain locks (e.g. file byte ranges), and capabilities (e.g. to cache file data locally). For instance in Ceph clients and metadata servers use a cooperative protocol in which clients voluntarily release resources back to the file system metadata service in order to implement sharing policies that are controlled by the metadata service.

In Ceph the locking service implements a capability-based system that expresses what data and metadata clients are allowed to access as well as what state they may cache and modify locally. While designed for the file abstraction, internal services such as indexing, locking, and caching are all common services that are useful to a broad spectrum of applications when considered in a more general form. For example, distributed applications often make use of locking services to control access to shared resources using external systems like Zookeeper. Next we examine a concrete example to show how these internal services can be repurposed to manage non-POSIX file shared resources by constructing the CORFU sequencer abstraction.

5.5.1 Example: CORFU sequencer

Recall from Section 4.7 that the CORFU log interface depends on a service called a sequencer for achieving high-performance. A sequencer service assigns log positions to clients by reading from a volatile, in-memory counter which can run at a very high throughput and at low latency. In Section 4.7 we showed how the durability of object data managed by Ceph could be changed to create an optimized data interface that implements the CORFU sequencer service. In this section we show how the file system metadata service can be repurposed to provide naming, and to implement the sequencer service.

The sequencer resource supports the ability to *read()* the current tail value

and get the *next()* position in the log which also atomically increments the tail position that can be stored as a single integer. We have implemented the sequencer service as a file type interface in Ceph. Compared to a solution that uses a programmable data interface, this approach has the added benefit of allowing the metadata service to handle naming, by representing each sequencer instance in the standard file system hierarchical namespace. An implementation of the sequencer service as we have seen in previous chapters is in large part conceptually simple to implement correctly; as a centralized service, it is trivial to maintain serialization over the metadata using an atomic increment operation. The primary challenge in mapping the sequencer resource to the metadata service is handling serialization correctly to maintain the global ordering provided by the CORFU protocol since we are now using a distributed service rather than a centralized one to build the sequencer interface.

Initially we sought to directly model the sequencer service in Ceph as a non-exclusive, non-cacheable resource within the metadata service, forcing clients to perform a round-trip access to the resource physically stored at the authoritative metadata server for the sequencer inode. This was our intuitive idea of how the internal protocols for managing shared state would function, providing a mechanism for implementing the sequencer semantics in a manner similar to that of using a centralized service architecture. Interestingly, we found that the capability system in Ceph attempts to reduce metadata service load by allowing clients that open a shared file to temporarily obtain an exclusive cached copy of the resource, resulting in a round-robin, best-effort batching behavior. The Ceph capability sub-system issues tokens to clients granting them a temporary capability. When a single client is accessing the sequencer resource it is able to increment the sequencer locally, reporting the updated state to the metadata service when

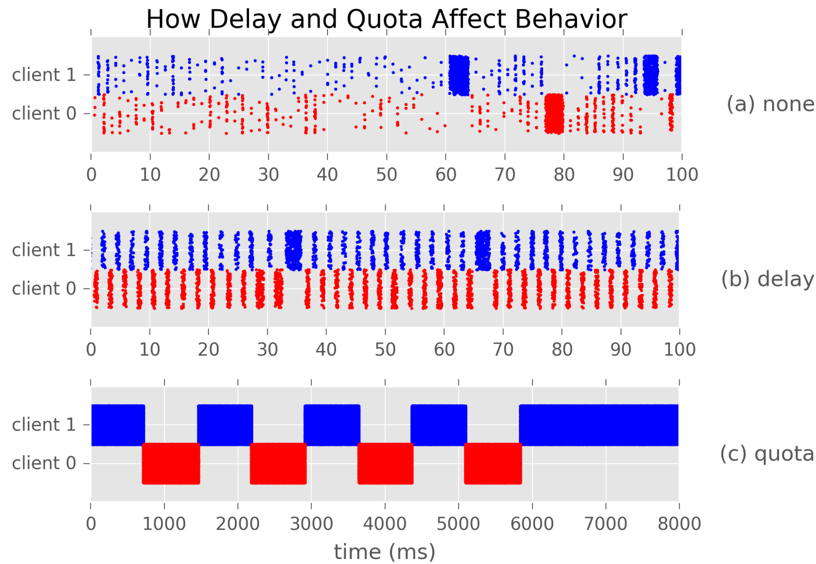


Figure 5.4: Each dot is an individual request, spread randomly along the y axis. The default behavior is unpredictable, “delay" lets clients hold the lease longer, and “quota" gives clients the lease for a number of operations.

its token is revoked. Any competing client cannot query the sequencer until the metadata service has granted it access and provided it with an up-to-date view of the sequencer state.

While unexpected, this discovery allowed us to explore an implementation strategy that we had not previously considered. In particular, for bursty clients, and clients that can tolerate higher latency, this mode of operation may allow a system to achieve much higher throughput than a system with a centralized sequencer service, given that clients can locally increment the state at very high rates that far exceed any network-based solution. We utilize the programmability of the metadata service to define a new policy for handling capabilities that controls the amount of time that clients are able to locally cache and control the sequencer resource. This allows an administrator or application more control beyond the standard best-effort policy that is present in Ceph by default.

The sequencer is implemented using as a new file type such that the sequencer

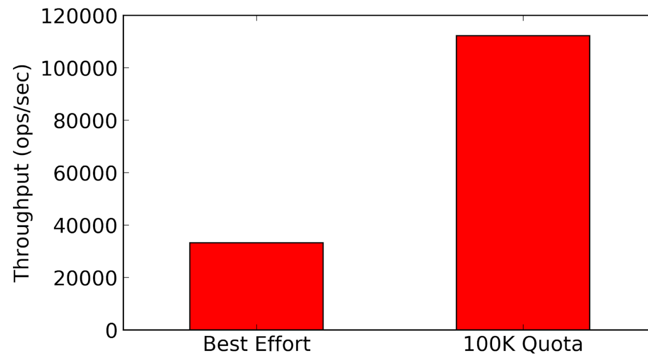


Figure 5.5: Throughput of best-effort and 100K quota policies. The best-effort policy spends more time communicating the capability between clients.

state (a 64-bit integer) is embedded in the inode of a file name. A total ordering of the sequencer state is imposed by the re-use of the file system capability service that can be used to grant exclusive access of inode state to clients. The metadata service is responsible for maintaining exclusivity and granting access. Figure 5.4 (a) shows the behavior of the system in which a best-effort policy is used. The two colors represent points in time that two clients were able to access the sequencer resource. The best-effort policy shows a high degree of interleaving between clients as exclusive access is shared, but the system spends a large portion of time re-distributing the capability, reducing overall throughput.

In order to control the performance of the system we implement a policy that (1) restricts the length of time that a client may maintain exclusive access and (2) limits the number of log positions that a client may generate without yielding to other clients waiting for access. The behavior of these two modes is illustrated in Figures 5.4 as (b) delay and (c) quota, respectively. Compared to the best-effort approach, using a delay-based policy can help tune the system to provide more balanced progress across clients. And a quota-based system can be tuned based on the expected burst size of applications that are not sensitive to longer delays involved in receiving log positions. Other policies such as combining quota and

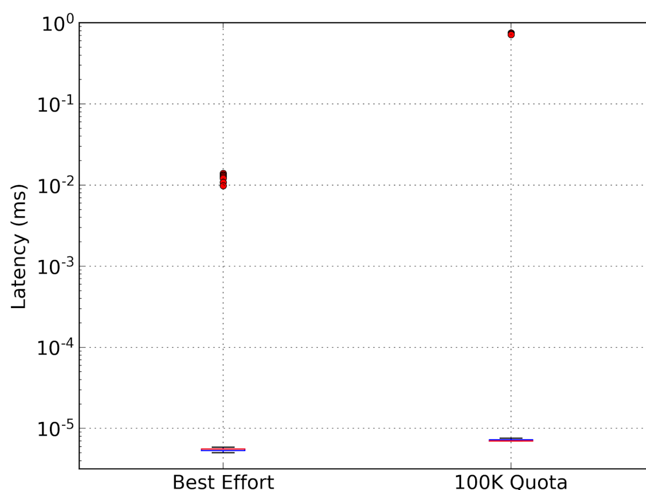


Figure 5.6: The latency of acquiring new log positions using a best-effort and quota-based capability policy. Latency is generally lower with the best-effort policy because the system attempts to balance the capability across clients based on time.

delay are also possible.

Figure 5.5 shows the throughput in log position assignments per second using the best-effort policy and a policy that enforces a 100K position quota. The best-effort policy is not able to provide throughput as high as the quota-based solution because the best-effort policy spends more time passing around the capabilities and this overhead leads directly to reduced throughput. However, as shown in Figure 5.6 the best-effort policy provides better latency and latency bounds for clients. The 100K quota experiment has several outliers of nearly 700 milliseconds, while the median of the best-effort is still lower than the quota-based policy. This is largely expected as the best-effort policy makes more of an effort to balance the capability across clients. The exact cause of the outliers is unknown, but may be caused by forcing the system into states that it does not normally operate in (e.g. a quota based policy), leading to unexpected behavior.

Recovery

Recall from Section 4.7 in which a reduced durability data interface was used to construct a service that implemented the CORFU sequencer interface. Since the sequencer state was stored in non-persistent DRAM, a failure of an OSD acting as the primary for the service resulted in a CORFU-specific recovery protocol being run to re-initialize the sequencer state of the OSD elected as the new primary. A similar but slightly more complex situation exists when the sequencer is constructed using the metadata capabilities service.

In a centralized version of the sequencer following a failure a new sequencer is elected and its state initialized from the result of the recovery protocol involving contacting each log storage target. One interesting aspect of the use of the capabilities service is that exclusive access to the sequencer state shared resource is implemented in part by physically moving the authoritative version of the data to the client. Thus, if a metadata service *or a client* fails while holding the authoritative version of the sequencer state then the metadata service must initiate the CORFU sequencer recovery protocol to obtain a new initialization state. The fact that clients may fail while controlling the authoritative copy of the sequencer state may have an impact on when such a service is deployed. This is because the capability subsystem depends on timeout mechanisms to make a decision that a client has failed and initiate recovery. Thus, a client that is malicious or not stable can easily reduce the overall performance of the system by preventing the updated sequencer state from being propagated back to the metadata service.

Future work

The most interesting piece of future work is to fully evaluate the performance profile that can be achieved by reusing the capabilities system for managing access

to shared resources. We have shown that different policies can be built that offer trade-offs, but it is left to future work to determine exactly when a sequencer that uses a solution that we've described in this section should be selected over a solution using a centralized sequencer. We would also like to apply this technique in other application domains and for other types of system resources.

5.6 Service metadata

Keeping track of state in a distributed system is an essential part of any service and a necessary component in order to diagnose and detect failures, when they occur. Service metadata is high-level information about the daemons in the system and includes things like cluster membership details, hardware layout (e.g., racks, power supplies, etc.), data layout, and daemon state and configuration. It differs from traditional file system metadata which is fine-grained information about files that applications control. While distributed file systems may manage millions of files that are frequently updated by multiple applications, service-level metadata is generally far less frequently updated and is smaller in volume. A simple example of service-level metadata is the registry of cluster membership used by Ceph clients to locate storage nodes; this list must be kept up-to-date on all clients, and remain consistent.

Currently in Ceph a consistent view of cluster state among server daemons and clients is critical to provide strong consistency guarantees to clients. Ceph maintains cluster state information in per-subsystem data structures called "maps" that record membership and status information. A Paxos [75] monitoring service is responsible for integrating state changes into cluster maps, responding to requests from out-of-date clients and synchronizing members of the cluster whenever there is a change in a map so that they all observe the same system state. As a

fundamental building block of many system designs, consensus abstractions such as Paxos are a common technique for maintaining consistent data versions, and are a useful system to expose.

The default behavior of the monitoring service in Ceph can be seen as a Paxos-based notification system, similar to the one introduced in [30], allowing clients to identify when new values (termed epochs in Ceph) are associated to given maps. Ceph does not expose this service directly, however we have found that domain-specific services implemented with programmability techniques have benefited from this being exposed as a key-value service designed for managing service metadata that is built on top of the consensus engine. Since the monitor is intended to be out of the high-performance I/O path, a general guideline is to make use of this functionality infrequently and to assign small values to maps.

While there are many different data models and update semantics that could be used for different applications, we have found it to be sufficient to expose expose a strongly-consistent view of time-varying service metadata as an interface rather than a hidden internal component. For example, this can be used to store object interfaces as well as metadata service load balancing policies [99]. The system provides a generic API for adding arbitrary values to existing subsystem cluster maps. As a consequence of this, applications can define simple but useful service-specific logic to the strongly-consistent interface, such as authorization control (only specific clients can write new values) or triggering actions based on specific values (e.g. sanitize values). Domain-specific interfaces and services make use of this functionality to register, version, and propagate dynamic code and configurations (e.g. Lua scripts) for new object interfaces defined in storage daemons and policies in the metadata load balancer. Using this service guarantees that interface definitions are not only made durable, but are transparently

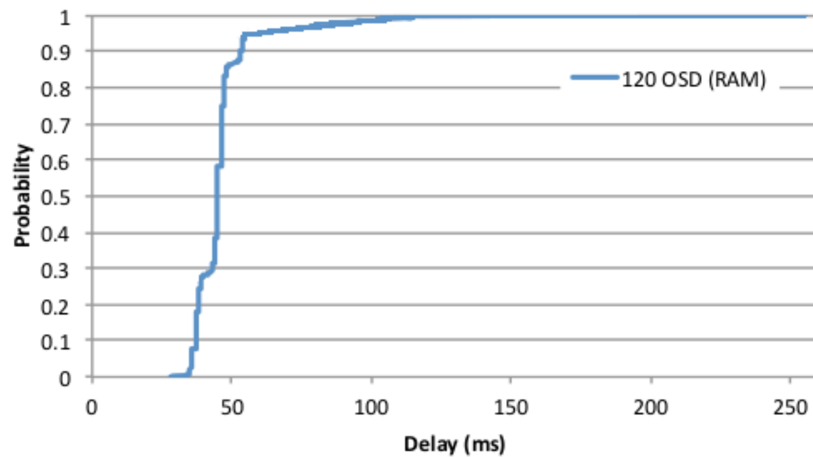


Figure 5.7: Cluster-wide interface update latency, excluding the Paxos proposal cost for committing the service metadata interface.

and consistently propagated throughout the cluster so that clients are properly synchronized with the latest version of an interface.

5.6.1 Interface propagation

The customization of storage system behavior and instantiation of domain-specific interfaces using programmability techniques inherently requires that state related to the interface to be propagated throughout the system. For example, installation of a new data interface in Ceph requires interface implementations to be installed on the storage devices in the system. This is true independent of the path by which the system arrives at a target implementation, such as by an application directly expressing semantics through executable scripts like Lua, or through code and execution plan generation using an optimizer and declarative programming techniques.

In a large cluster with many clients, one concern with programmability is the latency required to propagate changes throughout the system. This is particularly of concern for high-availability applications with low-latency requirements. For

example, a database management system may want to construct an interface custom tailored to a particular table that it manages, and be able to provide a service based on the interface to clients interactively. In general this means that a new interface must be installed on all nodes in the system. To this end we evaluate the performance of supporting the evolution of interfaces through automatic system-wide versioning and installation using the service metadata interface discussed in the previous section.

We demonstrate the feasibility of utilizing the Ceph monitoring sub-system by evaluating the performance of installing and distributing interface updates. Figure 5.7 shows the CDF of the latency of interface updates. The interfaces are Lua scripts embedded in the cluster map and distributed using a peer-to-peer gossip protocol. The latency is defined as the elapsed time following the Paxos proposal for an interface update until each object storage daemon makes the update live (the cost of the Paxos proposal is configurable and is discussed below). The latency measurements were taken on the nodes running object server daemons, and thus exclude the client round-trip cost. In each of the experiments 1000 interface updates were observed.

Figure 5.7 shows the lower bound cost for updates in a large cluster. In the experiment a cluster of 120 object storage daemons (OSDs) using an in-memory data store were deployed, showing a latency of less than 54 ms with a probability of 90% and a worst case latency of 194 ms. These costs demonstrate the penalty of distributing the interface in a large cluster. In practice the costs include, in addition to cluster-wide propagation of interface updates, the network round-trip to the interface management service, the Paxos commit protocol itself, and other factors such as system load. By default Paxos proposals occur periodically with a 1 second interval in order to accumulate updates. In a minimum, realistic quorum

of 3 monitors using hard drive-based storage, we were able to decrease this interval to an average of 222 ms.

5.7 Declarative approaches

Our investigation of programmability of metadata management services in Ceph revealed a number of opportunities to use declarative specification approaches. In Section 5.3 we showed how the file system namespace could be used to represent application-level non-file objects and the relationships between them in the case of streams contained with the CORFU log abstraction. For this use case, both the structure of the namespace as well as the interface are application-specific. Declarative specifications that act as rules on how a namespace is structured is useful for applications to ensure a consistent behavior. Declarative specifications for interfaces can be handled in the same way that data interfaces are handled. A critical use case of declarative specifications come from the challenges presented by portability. For example, a system like Ceph provides two high-level mechanisms for approaching the creation of the CORFU sequencer service, and the decision may be made by an optimizer according to a particular cost model and the needs of an application. However, it is not guaranteed that a non-Ceph storage system will contain the same set of mechanisms, or the same cost trade-offs. For instance, the Hadoop file system is optimized for single-writer cases and does not provide full POSIX consistency semantics. Therefore it is unlikely to have sophisticated capabilities mechanisms internally for managing cache coherency across clients. A declarative specification can be used to mask these changes from application developers by finding different mappings for services for a target storage system.

Each of the use cases we saw in this section will also benefit from using cost

models to inform behavior and implementation. For example, the size of metadata associated with an inode will effect scalability and performance depending on the capabilities of the underlying system, as well as the strategy of metadata management scaling used in a particular system. In Section 5.5.1 we showed that there are a variety of policies that can be used to control the performance of the capabilities-based CORFU sequencer, and the cost model for these policies must be weighed against each, and against entirely different approaches such as centralized approaches using data interface (see Section 4.7) which will have a distinctly different cost model and set of trade-offs.

5.8 Conclusion

It's not an exaggeration to say that a storage system is inherently a large metadata management system. Every piece of data in a storage system exists in support of managing and securing chunks of application-level data. To that end, storage systems contain a dizzying array of metadata management facilities to cover a variety of management scenarios both for internal data as well as application-level metadata. In this chapter we have examined two common metadata management services found in distributed storage systems and how programmability techniques can be applied to generalize these services for use by applications to create new features, and reduce the duplication of services.

The most prominent form of metadata management found in distributed storage systems is a POSIX file system. Such interfaces expose a bytestream interface to applications that allow them to organize file resources into a hierarchical namespace, and rely on the system to provide (in many cases) well-defined semantics on how data behaves when accessed concurrently. Unfortunately, even though applications store a very rich variety of data, a file system interface is limited

to managing only opaque byte streams. This forces applications to map arbitrary data models and management needs onto a single interface that may require complex middleware solutions and challenges in enabling scalability.

To address this we have introduced a file type abstraction that exposes the underlying scalable namespace for common naming services required by applications, and allow applications to programmability define new file types that expose domain-specific interfaces, rather than forcing the use of a POSIX file interface. The file type interface allows applications to define how new data formats should be read and written, allow applications to have access to a scalable naming service without the overhead of also managing files, as well as to take advantage of internal services that exist in support of the file system in application-specific ways. We demonstrated this by exposing the locking and capabilities sub-systems used by the Ceph file system for managing file cache coherency among a set of distributed clients. In particular we showed how the capability system could be used to create a new file type that implemented the interface of the CORFU sequencer, allowing the file system to act as both a service directory as well as the service itself. This was accomplished by defining the sequencer counter state in the file system inode, and reusing the capabilities interface to enforce atomic updates on the counter. The use of the capability system proved to provide a different performance profile than a traditional centralized system architecture, and in particular it has shown to be good for clients that have large burst rates.

Finally we examined an internal system in Ceph used to manage cluster-level metadata that includes things like cluster membership, the physical organization of hardware (e.g. racks, rooms), as well cluster-level abstractions such as the *pool* found in Ceph which is effectively a namespace of data that can be controlled as a distinct unit of administration from other pools. Systems for managing this type

of data are often implemented and used internally using a consensus engine such as Paxos. However, services such as Zookeeper (similar to a Paxos service) are often deployed alongside storage systems to manage application-level metadata such as application membership, and locking, leading to duplication of similar services. We exposed the internal consensus service found in Ceph to support consistent, cluster-wide distribution and installation of new data interfaces in support of storage system programmability. We demonstrated this by showing that new data interface definitions could be propagated through the peer-to-peer distribution system found in Ceph across 120 servers in less than 300 milliseconds, allowing fast turn around time for certain applications such as database management systems that use data interfaces to create data model specific optimizations.

Chapter 6

Development environment

This thesis has introduced programmable storage as a means by which internal storage abstractions can be generalized and reused to support the creation of new application-specific storage interfaces and services. This approach was demonstrated across several classes of applications using a variety of storage subsystems, but programmable storage contained several serious concerns related to design, software maintenance, and portability of new interfaces. In Section 2.4 we introduced declarative storage which replaced an ad hoc approach to building interfaces with declarative specifications, and relied on interface implementation generation using techniques from the database systems community. While an approach based on declarative specifications can capture the semantics of new storage interfaces and allow developers to ignore low-level details about internal interfaces, we have not addressed some real-world concerns related to the development experience of a programmable storage system.

The challenge stems of the inherent ability to create storage interfaces, independent of the method by which they are created (e.g. through a declarative specification). Storage interfaces and data management systems in general are inextricably tied to the ability to interpret and access data. This effectively ele-

vates the criticality of the preservation and management of interfaces in storage systems to that of data artifacts themselves. However, there are currently no services in storage systems for managing the life cycle of application-driven, dynamic interface development.

Because storage interfaces are often tightly coupled with applications and services, a process of co-design implies that data interface development can be closely related to the development of higher level services. In particular, it is very common for engineering teams to follow a branch-and-merge source-code management style using software such as Git or Subversion, in which feature branches are merged into a production line after some period of insulated feature development and maturation. While application feature development can often take place using, for example small-scale deployments on developer desktops, the same is not true for storage system interface development, where access to distributed resources and the peculiarities of live data are crucial to feature development and testing correctness at scale. One option is to allow developers unconstrained access to the storage system, relying on informal, error prone team guidelines to avoid conflicts such as naming or data format incompatibilities. Yet another option would be to maintain a smaller development cluster, but this leads to increased costs and may not expose the development process to realistic conditions. It would be useful if a storage system provided a development environment for storage interfaces as a first class service akin to the isolated development workflows for application developers using source-code management tools.

6.1 Overview

In the remainder of this chapter we propose a solution based on the concept of a developer *workspace*. A workspace represents a unit of isolation within the storage

system that allows for the independent evolution of interfaces that are dynamically created by application developers. We argue that due to the nature of interfaces and their need for preservation, as well as the isolation needs of applications for security and performance, that workspace and the data interfaces defined within them, should be fully managed by the storage system. Finally we enumerate several key areas of work that need to be addressed. These include handling data isolation between interfaces, creating workflows for application developers to resolve conflicts, handling the evolution of interfaces including interface version management, and performance isolation concerns.

6.2 Motivation

We use a generic data analysis application as motivation in this chapter, but any other examples and use cases described in this thesis would be applicable as well. The collection and analysis of large-scale read-mostly data such as access logs, click streams, and sensor data, as well as scientific simulation output, require scalable, fault-tolerant storage systems. Figure 6.1 illustrates a typical architecture in which data, such as time-ordered logs, are partitioned by attributes such as time or data source, and stored within objects in a distributed object store such as Ceph. Shown in the same figure is a production application that interacts with the stored data objects by remotely reading and producing analysis results, or searching for activity patterns. This is not unlike previous examples and use cases described in Chapter 4 in which domain-specific interfaces provide access to storage system resources to enable features like filtering and push-down aggregates.

Now consider the following possible mode of development: engineering team(s) may be developing and testing new features such as new filtering algorithms,

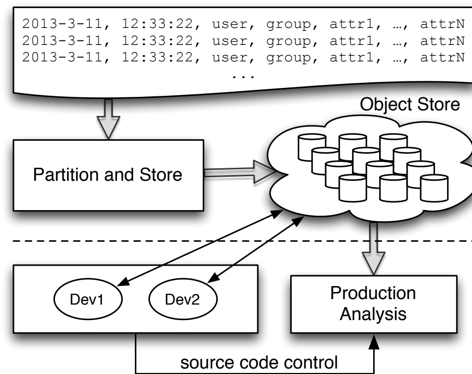


Figure 6.1: Log data is stored in objects that are batch analyzed while developers create new features and evolve the system.

and simultaneously evolving a production deployment using standard source-code management techniques, workflows, and deployment procedures. In the context of a programmable storage system, an open question is how much of the management responsibility of data interfaces is placed on applications compared to the storage system itself.

Throughout this thesis it has been assumed that a storage system is largely dedicated to one user or application during development. Furthermore, we have largely made no assumptions about how code or specifications that define new interfaces are stored and managed. Since applications are developed using source-code control systems, but no such abstractions exist in the storage system, applications benefit from explicitly managing storage interfaces. However, as we have seen, the creation of implementations from declarative specifications is tightly coupled to the storage system, suggesting that the storage system plays a critical role in managing interfaces.

Beyond managing interface specifications, there are other reasons for interface development to be closely integrated with the storage system. Development of interfaces in practice may depend on the datasets that are present in production systems, or depend on the nuances of large scale systems for testing. In order

avoid moving large amounts of data or depending on provisioning hardware for development clusters, it would be desirable for the storage system to natively support development processes. In addition to practical issues such as performance and security isolation, one reason this is difficult is because there is a tight coupling between storage interfaces and applications that require both to be able to *evolve together* through a standard software development life cycle. Instead of applications being fully responsible for interface management and development, managing the deployment, consistency, and versioning of interfaces, as well as enforcing isolation between developers and production interfaces, is a task best handled by the storage system itself as a shared resource and service.

6.2.1 Storage interface evolution

Dynamically created storage interfaces pose a challenge for software development because application software may evolve independently from the deployed storage interfaces, but still require strong version consistency and compatibility between the application and deployed interfaces. Additionally, recall from Figure 6.1 that multiple developers with common developer workflows may evolve an application by first developing and testing features, then integrating the changes into a production deployment. In order for each developer to work on features independently, conflicts that result from customized interfaces must be isolated and managed.

Consider the application life cycle depicted in Figure 6.2. Developers *Dev1* and *Dev2* are responsible for developing independent, domain-specific interfaces to individual objects (e.g. arithmetic mean, and minimum, as shown) that will replace the same per-object operation performed remotely by the analysis application. This scenario is analogous to the use of application-specific data interfaces

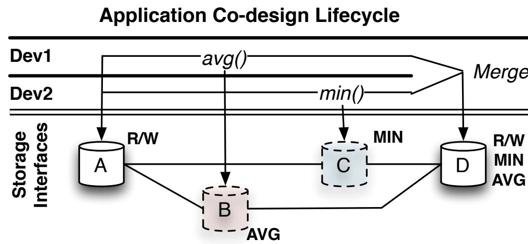


Figure 6.2: Developers evolve application software and storage interfaces through a co-design process.

to accelerate query performance in Skyhook as previously discussed in Section 4.5. In a multi-developer environment, each developer must now evolve the storage-level interfaces, as well as change application-level code to take advantage of the new features, all while ensuring that the application and storage interfaces remain synchronized. For instance, both developers begin with a base storage interface exposing the standard byte-oriented interface (ver. A). Each developer evolves the application and storage interfaces with their respective features (ver. B, C). Once the features are complete, they are merged to expose the new interfaces to the production application (ver. D). Two interfaces can conflict if local object resources are not partitioned. For instance, if two interfaces implementing distinct statistical calculations (e.g. *mean* vs *median*) cache their result in a local object attribute to avoid recomputation, but use the same attribute name (e.g. *avg*), data corruption may lead to silent errors and unexpected results. Thus, providing transparent isolation between interfaces is important in order to avoid the type of ad hoc coordination among developers that would otherwise be required such as using naming conventions and other techniques that are prone to user error.

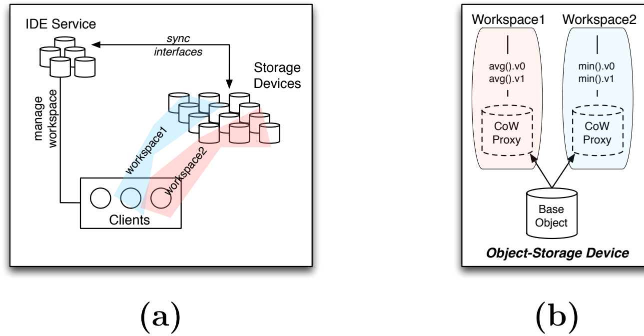
6.3 Storage development workspace

In order to manage developers in a programmable storage system we propose adopting abstractions similar to those already in use by developers. Specifically, we propose an *interface developer environment* (IDE) for constructing new storage interfaces that centers around the use of an isolated *workspace* abstraction that is well-aligned to common software development workflows, and integrates with existing approaches to developement using programmability techniques. We don't make any assumptions about the mode of programmability, which may be low-level (e.g. code injection), or high-level using proposed techniques based on declarative approaches to building storage interfaces (see Section 2.4).

6.3.1 Workspaces

A workspace is an entity managed by the storage system which provides isolation between storage interfaces. Workspaces can be created, destroyed, and merged through the use of the *interface development environment* (IDE) service, illustrated in Figure 6.3a. The IDE service exposes an interface similar to that of Git or Subversion in which a development branch forms the basic unit of isolation. It is expected that the use of a workspace will resemble a developer's *working copy* in the traditional sense of source-code control systems, providing a safe environment to construct, test, and refine a line of feature development. The key difference being that a workspace exists within and is managed entirely by the storage system for the purpose of developing new storage interfaces.

Isolation. Providing efficient isolation among workspaces is a primary challenge. In order to avoid expensive data duplication, the system should allow interfaces to share as much data as possible. For instance, read operations performed by an interface should require no special handling, and be satisfied by



(a) (b)

Figure 6.3: In (a) clients use an IDE service to create workspaces that form a context within the storage system. In (b) base data is not duplicated, and CoW provides isolation for interface private data.

base data. However, writes must be carefully handled as to not interfere with state created by interfaces in other workspaces, or byte stream data associated with the object. For instance, a data interface that caches computed data using an internal service such as the omap key-value store must take care not to select keys that conflict with other interface implementations (e.g. two interface caching a value named generically as *average*). As a shared resource managing access by independent clients, the storage system is in the position to not only enforce this requirement but also manage it by providing isolation. Efficient techniques for isolation depend on the type of service. For instance, isolation can be achieved efficiently using namespace techniques when storing key/value pairs, while copy-on-write techniques can be used for data transformations on large binary data.

Performance isolation is also an important aspect to consider. Given the large body of work on performance management and isolation in storage systems, we expect that existing techniques will be applicable in this context. For example, many techniques enforce isolation on the basis of a generic abstraction such as a *stream* or group of streams. Since a workspace is only a logical grouping resources and abstractions like streams, existing techniques may be able to be applied. Fully evaluating existing techniques and their applicability to this context is future

work. Additionally, a higher level abstraction like a workspace may even enable new forms of control and isolation.

Partitioning. While logical isolation is important to ensure correctness, an organization may want to physically partition its storage in such a way that development workspaces reside on distinct hardware. The logical workspace entity should integrate with existing facilities within the storage system for custom data placement and tiering policies, allowing subsets of data to be placed onto specific sets of nodes. Workspaces can be linked to these physical partitions through existing system abstractions (e.g. a *pool* in Ceph) which ensure that the space of addressed objects is constrained by the physical partitioning. This is also useful as a means of forcing isolation through physical partitioning.

6.3.2 Workspace management

Ultimately, interfaces defined within workspaces as part of application development will be migrated into a production environment. For instance, the interfaces defined in separate workspaces shown in Figure 6.2 can be merged into production, providing access to the union of the interfaces to applications accessing the storage system in the context of the production workspace.

There are several issues that may arise when merging workspaces. First, at a high-level merging changes the visibility of interfaces, and as a result interface naming conflicts may arise. For instance, two workspaces may define the same interface. These types of conflicts are largely application-specific and must be handled explicitly by developers. Like source-code management systems, the primary responsibility of the storage system is to provide feedback to developers about the changes they are making through the interface development service, and help manage conflict resolution.

Interfaces that utilize private data can be merged without low-level conflicts by migrating the same isolation parameters (e.g. namespacing) used to prevent conflicts between workspaces. However, for interfaces that perform heavy-weight data transforms such as using new data layouts, migrating all interfaces to a use a new layout may be necessary. In order to make format migration easier, workspace merging should optionally specify a transformation routine that the system ensures is applied prior to invoking any interface following a merge operation. Finally, the removal of workspaces will result in lazy deletion of all unmerged interface state created during the lifetime of the workspace.

6.4 The IDE Service

Interfaces and workspaces are cluster-wide entities that must be managed by the storage system in the face of cluster failures, expansion, and policy changes. For instance, the storage system must ensure that a new storage node has the required interface specifications present before it can service requests from applications requiring these interfaces. Further, newly registered versions of interfaces must be propagated to nodes within the system, and properly synchronized with applications expecting the latest version.

Luckily, existing services within distributed storage systems solve similar problems. For instance, a core service often found in distributed systems is a highly available versioned data store commonly implemented using a consensus algorithm, such as Paxos. For instance, Ceph uses *monitor services*, built upon Paxos, to manage cluster membership, service discovery, replicated logs, and authentication. In Section 5.6.1 we showed how this service could be reused to propagate a consistent view of interface definitions throughout a cluster. That was a specific instance of programmability being applied to managing cluster-wide metadata. A

similar approach can be used to create an application-specific management solution for cluster-wide metadata involving the versioning of interface definitions.

Finally, a mechanism is needed to associate interface versions managed by the storage system with the versions that applications expect. We are considering two possible solutions to this problem. First, some source-code control systems provide the ability to inject external context information into the managed content (e.g. CVS tags expansion). Extending or exploiting this feature may allow us to automatically generate version macros used to provide context when clients access the system. Similarly, systems such as Git allow external repositories to be seamlessly integrated into existing repositories. By allowing the storage system to export its own virtual Git repository, we can enable the system to present previously registered, versioned code automatically into a higher-level project repository. Providing an easy-to-use and robust integration solution is important for usability, and utilizing other techniques from RPC stub generation may prove to be valuable.

6.5 Summary

The technical aspects discussed in this thesis, from ad hoc development of domain-specific interfaces to declarative specifications to handle issues that arise from programmability, are only one part of a larger solution that is needed for adoption of programmability in storage systems. In order to create a system that is useable real-world challenges related to development, management, and preservation of interfaces need to be solved.

In this chapter we have proposed that storage systems take a first-class role in managing these development processes. First, we observe that interfaces are critical to being able to access and interpret stored data. As such, they must be

stored and treated as importantly as the data itself. Second, like application development storage interface development is an iterative and collaborative process. We propose that storage system expose a workspace abstraction that is analogous to the concept of a working copy used in modern software development workflows in which developers operate in an isolated environment and collaborate to merge lines of development. Since a workspace for managing storage interfaces necessarily requires the environment of the storage system, exposing an isolated workspace abstraction will require solutions to many challenges related to data and performance isolation, handling conflicts between interfaces, and interface preservation and versioning.

Chapter 7

Conclusion

We finish up by discussing what we would like to do in the future, and provide a short summary of the thesis.

7.1 Future work

The directions opened up for future work are substantial.

7.1.1 Programmable components and applications

We have really only scratched the surface when it comes to exploring how the internal components of the storage system can be used to support applications and new storage interfaces. For example, we were able significantly improve the performance of multi-client workloads of the CORFU log abstraction by allowing multiple log appends to be handled in the same transaction. The changes required for this involve domain-specific customizations to the queueing semantics in the storage server, and are a good candidate for programmability.

Ceph also contains several unique subsystems. The tiering sub-system in Ceph allow write-back and write-through policies to be attached to objects such that

data may be cached on a fast tier, with primary storage on a slower tier. Generalizations of tiering may be very useful to achieve various application-specific optimizations. For example, the Skyhook driving example described in Section 4.5 could utilize much of the mechanisms in the tiering subsystem to cache relational table data stored in multiple formats, each optimized for a particular workload. Another unique subsystem found in Ceph is a publish-subscribe notification system that can send messages triggered by actions on objects. These actions and the object communication mechanisms found in the tiering subsystem may be combined into various data flows. When combined with application specific data interfaces, several distributed processing patterns can be realized.

We have greatly benefited from having a large set of driving applications and data services. There is significant opportunity for a depth-first approach to further integrating the use cases in this thesis, but we would also like to expand the set of applications and data interfaces that can benefit. Expanding the set of applications is also useful for driving the identification of new subsystems that can be generalized and used to support storage programmability.

7.1.2 Additional systems

Currently developers are accepting Ceph as the primary system for building new data interfaces. And some of the challenges that we have discussed that motivated declarative storage are relevant even when restricted to Ceph. However, without the ability to move between systems, system lock-in in still a fear; even though the system can be changed, the more that changes rely on the single malleable system the harder it is to migrate. We have highlighted several additional systems and argued that these systems contains similar mixes of sub-systems. However, fully exploring this is important future work.

7.1.3 Declarative storage

The most important aspect of future work, and perhaps the most ambitious, is continuing the work on declarative storage. Showing that existing interfaces can be expressed declaratively using languages like Bloom has been important, but the next step is to show that an implementation of an interface can be generated automatically. This will require a deep collaboration between the database and storage system communities.

7.2 Summary

All indications point to a future where developers and organizations are comfortable with the development of domain-specific storage interfaces over existing standardized interfaces like POSIX file I/O. This is a drastic shift from the past, but the change is taking place right now. Currently this is happening as an ad hoc process primarily within the Ceph storage system which is widely deployed across industry. As the capability to build new interfaces is expanded to encompass a wider variety of application developers and ultimately other storage systems, standards and best-practices will emerge. In this thesis we examined this design space and developed proposals for new ways of building interfaces based on declarative specifications that address many of the issues and concerns with existing approaches.

First we motivated programmable storage by differentiating it from other approaches to building new storage system interfaces. We showed that by exposing storage system services that applications could avoid duplicating complex services and simplify system design. We then illustrated how storage programmability suffers from several real-world challenges including software maintainability and

portability, as well as a large design space that requires storage system expertise.

These issues motivated the proposal of building storage system interfaces using declarative languages. By specifying storage interfaces declaratively, the low-level implementation details that lead to the challenges present in an ad hoc approach were hidden from developers and could be handled using tools and techniques from the database systems community such as query execution plan generation and optimization. We used the CORFU distributed shared-log abstraction as a driving example and showed how its semantics could be expressed using an existing declarative language called Bloom.

We then explored a spectrum of internal services present in the Ceph storage system and showed how they could be generalized to support a variety of applications. We examined the creation of new data interfaces supporting applications by exposing internal services such as indexing, caching, and durability, as well exposing resources such as CPU, memory, and I/O bandwidth. We then repeated the process with metadata management subsystems, reusing components of the POSIX file system to implement aspects of the CORFU log abstraction, and the cluster-level Paxos service for managing interface specifications in support of programmability itself. Finally we addressed real-world concerns about the experience of developers interacting with a programmable storage system by proposing that the storage system expose a first-class workspace abstraction for isolating developer activity.

Bibliography

[1] Avro serialization format. `https://avro.apache.org/`.

[2] CruzDB. `https://github.com/cruzdb/cruzdb`.

[3] Gluster storage system. `https://www.gluster.org/`.

[4] Kinetic ethernet drives. `https://www.seagate.com`.

[5] Messagepack serialization format. `https://msgpack.org/index.html`.

[6] Openstack swift object storage. `https://wiki.openstack.org/wiki/Swift`.

[7] PLFS traces. `http://institutes.lanl.gov/plfs/maps/`.

[8] Protocol buffers serialization format. `https://developers.google.com/protocol-buffers/`.

[9] Paxos made simple. Leslie Lamport, November 1 2001.

[10] FastForward storage and I/O stack design documents. `https://wiki.hpdd.intel.com/display/PUB/Fast+Forward+Storage+and+IO+Program+Documents`, 2014.

[11] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. ACM.

[12] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[13] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *CIDR '11*, 2011.

[14] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. Lineage-driven fault injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 331–346, New York, NY, USA, 2015. ACM.

[15] Khalil Amiri, David Petrou, Gregory R. Ganger, and Garth A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '00, pages 25–25, Berkeley, CA, USA, 2000. USENIX Association.

[16] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. Disk-locality in datacenter computing considered irrelevant. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 12–12, Berkeley, CA, USA, 2011. USENIX Association.

[17] Mahesh Balakrishnan et al. Tango: Distributed data structures over a shared log. In *SOSP '13*, Farmington, PA, November 3-6 2013.

[18] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A shared log design for flash clusters. In *NSDI'12*, San Jose, CA, April 2012.

[19] Kyle Banker. *MongoDB in Action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[20] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 66:1–66:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[21] Babak Behzad, Joseph Huchette, Huong Vu Thanh Luu, Ruth Aydt, Surendra Byna, Yushu Yao, Quincey Koziol, and Prabhat. A framework for auto-tuning hdf5 applications. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 127–128, New York, NY, USA, 2013. ACM.

[22] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. PLFS: A Checkpoint Filesystem for Parallel Applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

[23] Phil Bernstein, Colin Reid, Ming Wu, and Xinhao Yuan. Optimistic concurrency control by melding trees. volume 4, pages 944–955, January 2011.

[24] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD '15*, 2015.

[25] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder – a transactional record manager for shared flash. In *CIDR '11*, Asilomar, CA, January 9-12 2011.

[26] Ceph blog. `https://ceph.com/geen-categorie/500-osd-ceph-cluster/`.

[27] Ceph blog. `https://ceph.com/community/new-luminous-scalability/`.

[28] Paul G Brown. Overview of scidb: large scale array storage, processing and analysis. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 963–968. ACM, 2010.

[29] Joe B. Buck, Noah Watkins, Jeff LeFevre, Kleoni Ioannidou, Carlos Maltzahn, Neoklis Polyzotis, and Scott Brandt. SciHadoop: array-based query processing in Hadoop. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11.

[30] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[31] Bradford L Chamberlain, David Callahan, and Hans P Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[32] Samy Chambi, Daniel Lemire, Owen Kaser, and Robert Godin. Better bitmap performance with roaring bitmaps. *Softw. Pract. Exper.*, 46(5):709–719, May 2016.

[33] C. Chen and Y. Chen. Dynamic active storage for high performance i/o. In *2012 41st International Conference on Parallel Processing*, pages 379–388, Sept 2012.

[34] C. Chen, Y. Chen, and P. C. Roth. Dosas: Mitigating the resource contention in active storage systems. In *2012 IEEE International Conference on Cluster Computing*, pages 164–172, Sept 2012.

[35] Steve Chiu, Wei-keng Liao, and Alok Choudhary. *Design and Evaluation of Distributed Smart Disk Architecture for I/O-Intensive Workloads*, pages 230–241. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[36] Mariano P. Consens, Kleoni Ioannidou, Jeff LeFevre, and Neoklis Polyzotis. Divergent physical design tuning for replicated databases. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 49–60, New York, NY, USA, 2012. ACM.

[37] Peter F Corbett and Dror G Feitelson. The vesta parallel file system. *ACM Transactions on Computer Systems (TOCS)*, 14(3):225–264, 1996.

[38] Skyhook database architecture. `https://sites.google.com/site/skyhookdb/`.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation*, OSDI '04.

[40] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[41] Phillip M. Dickens and Jeremy Logan. A high performance implementation of MPI-IO for a Lustre file system environment. 22.

[42] Jaeyoung Do, Yang-Suk Kee, Jignesh M Patel, Chanik Park, Kwanghyun Park, and David J DeWitt. Query processing on smart ssds: opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1221–1230. ACM, 2013.

[43] Matthieu Dorier. Composing hpc micro-services to build application-tailored distributed object stores. In *SIG-IO-UK Workshop '18 (Slides)*, 2018.

[44] Matthieu Dorier, Matthieu Dreher, Tom Peterka, and Robert B. Ross. Coss: proposing a contract-based storage system for HPC. In *PDSW 17'*, 2017.

[45] D. H. C. Du. Intelligent storage for information retrieval. In *International Conference on Next Generation Web Services Practices (NWeSP'05)*, pages 7 pp.–, Aug 2005.

[46] E. J. Felix, K. Fox, K. Regimbal, and J. Nieplocha. Active storage processing in a parallel file system. In *6th LCI International Conference on Linux Clusters: The HPC Revolution*, Chapel Hill, NC, April 26 2005.

[47] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[48] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, AD '11, pages 36–47, New York, NY, USA, 2011. ACM.

[49] Roxana Geambasu, Amit A. Levy, Tadayoshi Kohno, Arvind Krishnamurthy, and Henry M. Levy. Comet: an active distributed key-value store. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10.

[50] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03*, Bolton Landing, NY, 2003. ACM.

[51] Christos Gkantsidis, Dimitrios Vytiniotis, Orion Hodson, Dushyanth Narayanan, Florin Dinu, and Antony Rowstron. Rhea: Automatic filtering for unstructured cloud storage. In *NSDI'13*, Lombard, IL, April 2-5 2013.

[52] Raúl Gracia-Tinedo, Josep Sampé, Edgar Zamora, Marc Sánchez-Artigas, Pedro García-López, Yosef Moatti, and Eran Rom. Crystal: Software-defined storage for multi-tenant object stores. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, pages 243–256. USENIX Association, 2017.

[53] Matthias Grawinkel, Tim Süß, Gregor Best, Ivan Popov, and Andre Brinkmann. Towards dynamic scripted pnfs layouts. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 13–17. IEEE, 2012.

[54] Gary Grider. Personal correspondence. 2012.

[55] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface.* MIT Press, Cambridge, MA, USA, 1999.

[56] Matthew Haines, Piyush Mehrotra, and John Van Rosendale. *SmartFiles: An OO approach to data file interoperability*, volume 30. ACM, 1995.

[57] Jun He, John Bent, Aaron Torres, Gary Grider, Garth A. Gibson, Carlos Maltzahn, and Xian-He Sun. I/O acceleration with pattern detection. In *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13, New York, NY, USA - June 17 - 21, 2013*, pages 25–36, 2013.

[58] Yongqiang He, Rubao Lee, Yin Huai, Zheng Shao, Namit Jain, Xiaodong Zhang, and Zhiwei Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1199–1208. IEEE, 2011.

[59] James V Huber Jr, Andrew A Chien, Christopher L Elford, David S Blumenthal, and Daniel A Reed. Ppfs: A high performance portable parallel file system. In *Proceedings of the 9th international conference on Supercomputing*, pages 385–394. ACM, 1995.

[60] Larry Huston, Rahul Sukthankar, Rajiv Wickremesinghe, M. Satyanarayanan, Gregory R. Ganger, Erik Riedel, and Anastassia Ailamaki. Diamond: A storage architecture for early discard in interactive search. In *FAST 2004*, pages 73–86, San Francisco, CA, April 2004. USENIX.

[61] Roberto Ierusalimschy, Luiz Henrique De Figueiredo, and Waldemar Celes Filho. Lua-an extensible extension language. *Softw., Pract. Exper.*, 26(6):635–652, 1996.

[62] L. Ionkov, M. Lang, and C. Maltzahn. Drepl: Optimizing access to application data for analysis and visualization. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–11, May 2013.

[63] Florin Isaila and Walter F Tichy. Clusterfile: A flexible physical layout parallel file system. *Concurrency and Computation: Practice and Experience*, 15(7-8):653–679, 2003.

[64] M. Isenburg, P. Lindstrom, and H. Childs. Parallel and streaming generation of ghost data for structured grids. *IEEE Computer Graphics and Applications*, 30(3):32–44, May 2010.

[65] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: intelligent distributed storage. *Proceedings of the VLDB Endowment*, 10(11):1202–1213, 2017.

[66] Insoon Jo, Duck-Ho Bae, Andre S Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel DG Lee, and Jaeheon Jeong. Yoursql: a high-performance database system leveraging in-storage computing. *Proceedings of the VLDB Endowment*, 9(12):924–935, 2016.

[67] T. M. John, A. T. Ramani, and J. A. Chandy. Active storage using object-based devices. In *2008 IEEE International Conference on Cluster Computing*, pages 472–478, Sept 2008.

[68] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[69] John F Karpovich, Andrew S Grimshaw, and James C French. Extensible file system (elfs): an object-oriented approach to high performance file i/o. In *ACM SIGPLAN Notices*, volume 29, pages 191–204. ACM, 1994.

[70] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, September 1998.

[71] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie. Quantifying the energy cost of data movement in scientific applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 56–65, Sept 2013.

[72] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Reflex: Remote flash ≈ local flash. *SIGPLAN Not.*, 52(4):345–359, April 2017.

[73] Rusty Klophaus. Riak core: Building distributed applications without shared state. In *ACM SIGPLAN Commercial Users of Functional Programming*, CUFP '10, pages 14:1–14:1, New York, NY, USA, 2010. ACM.

[74] David Kotz. Multiprocessor file system interfaces. In *Parallel and Distributed Information Systems, 1993., Proceedings of the Second International Conference on*, pages 194–201. IEEE, 1993.

[75] Leslie Lamport. Paxos made simple. 32.

[76] H. Lim, V. Kapoor, C. Wighe, and D. H. c. Du. Active disk file system : A distributed, scalable file system. In *2001 Eighteenth IEEE Symposium on Mass Storage Systems and Technologies*, pages 101–101, April 2001.

[77] Jay Lofstead, Ivo Jimenez, Carlos Maltzahn, Quincey Koziol, John Bent, and Eric Barton. Daos and friends: a proposal for an exascale storage system. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 50. IEEE Press, 2016.

[78] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, CLADE '08.

[79] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Trans. Storage*, 13(1):5:1–5:28, March 2017.

[80] Lua. `https://www.lua.org`.

[81] John MacCormick, Nick Murphy, Marc Najork, andramohan A. Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage frastructure. San Francisco, CA, December 2004.

[82] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A Thekkath, and Lidong Zhou. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI*, volume 4, pages 8–8, 2004.

[83] Adam Manzanares, John Bent, Meghan Wingate, and Garth A. Gibson. The power and challenges of transformative I/O. In *2012 IEEE International Conference on Cluster Computing, CLUSTER 2012, Beijing, China, September 24-28, 2012*, pages 144–154, 2012.

[84] David Nagle, Denis Serenyi, and Abbie Matthews. The panasas activescale storage cluster: Delivering scalable high bandwidth storage. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 53. IEEE Computer Society, 2004.

[85] Nils Nieuwejaar and David Kotz. The galley parallel file system. *Parallel Computing*, 23(4-5):447–476, 1997.

[86] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[87] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[88] Juan Piernas and Jarek Nieplocha. Efficient management of complex striped files in active storage. In *Euro-Par 2008*, Gran Canaria, Spain, 2008.

[89] Lin Qiao, Vijayshankar Raman, Inderpal Narang, Prashant Pandey, David D. Chambliss, Gene Fuh, James A. Ruddy, Ying-Lin Chen, Kou-Horng Yang, and Fen-Ling Ling. Integration of server, storage and database

stack: Moving processing towards data. In *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 1200–1208, 2008.

[90] Simona Rabinovici-Cohen, Ealan Henis, John Marberg, and Kenneth Nagin. Storlet engine: performing computations in cloud storage.

[91] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

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

[93] M. T. Runde, W. G. Stevens, P. A. Wortman, and J. A. Chandy. An active storage framework for object storage devices. In *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12, April 2012.

[94] Salesforce. Personal correspondence. 2016.

[95] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[96] Sudharsan Seshadri, Mark Gahagan, Meenakshi Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *OSDI*, pages 67–80, 2014.

[97] M. Sevilla, I. Jimenez, N. Watkins, J. LeFevre, P. Alvaro, S. Finkelstein, P. Donnelly, and C. Maltzahn. Cudele: An api and framework for programmable consistency and durability in a global namespace. In *IPDPS '18*, 2018.

[98] M. Sevilla, R. Nasirigerdeh, C. Maltzahn, J. LeFevre, N. Watkins, P. Alvaro, M. Lawson, and J. Lofstead. Tintenfisch: File system namespace schemas and generators. In *HotStorage '18*, 2018.

[99] Michael A Sevilla, Noah Watkins, Ivo Jimenez, Peter Alvaro, Shel Finkelstein, Jeff LeFevre, and Carlos Maltzahn. Malacology: A programmable storage system. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 175–190. ACM, 2017.

[100] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *FAST*, pages 287–293, 2014.

[101] K. Shvachko, Hairong Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proceedings of the 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, 2010.

[102] Muthian Sivathanu, Andrea C. Arpaci-dusseau, and Remzi H. Arpaci-dusseau. Evolving rpc for active storage. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2002)*, pages 264–276, San Jose, CA, December 2002.

[103] Clinton Wills Smullen, IV, Shahrukh Rohinton Tarapore, Sudhanva Gurumurthi, Parthasarathy Ranganathan, and Mustafa Uysal. Active storage revisited: The case for power and performance benefits for unstructured data processing applications. In *Proceedings of the 5th Conference on Computing Frontiers*, CF '08, pages 293–304, New York, NY, USA, 2008. ACM.

[104] Seung Woo Son, Samuel Lang, Philip Carns, Robert Ross, Rajeev Thakur, Berkin Özisikyilmaz, Prabhat Kumar, Wei keng Liao, and Alok Choudhary. Enabling active storage on parallel i/o software stacks. In *MSST2010*, Incline Village, NV, May 3-7 2010.

[105] TPC Benchmark Specification. `http://www.tpc.org`.

[106] Ioan A Stefanovici, Bianca Schroeder, Greg O'Shea, and Eno Thereska. sroute: Treating the storage stack like a network. In *FAST*, pages 197–212, 2016.

[107] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Radu Stoica, Bernard Metzler, Nikolas Ioannou, and Ioannis Koltsidas. Crail: A high-performance i/o architecture for distributed data processing. *IEEE Data Eng. Bull.*, 40:38–49, 2017.

[108] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective i/o in romio. In *Proceedings of the The 7th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS '99, pages 182–, Washington, DC, USA, 1999. IEEE Computer Society.

[109] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: a software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196. ACM, 2013.

[110] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter J. Desnoyers, and Yan Solihin. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, FAST'13, pages 119–132, Berkeley, CA, USA, 2013. USENIX Association.

[111] Sean Treichler, Michael Bauer, and Alex Aiken. Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 263–276, New York, NY, USA, 2014. ACM.

[112] Mustafa Uysal, Anurag Acharya, and Joel H. Saltz. Evaluation of active disks for decision support databases. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, Toulouse, France, January 8-12, 2000*, pages 337–348, 2000.

[113] Noah Watkins, Zhihao Jia, Galen Shipman, Carlos Maltzahn, Alex Aiken, and Pat McCormick. Automatic and transparent i/o optimization with storage integrated application runtime support. In *Proceedings of the 10th Parallel Data Storage Workshop*, PDSW '15, pages 49–54, New York, NY, USA, 2015. ACM.

[114] Michael Wei, John D. Davis, Ted Wobber, Mahesh Balakrishnan, and Dahlia Malkhi. Beyond block I/O: implementing a distributed shared log in hardware. In *6th Annual International Systems and Storage Conference, SYSTOR '13, Haifa, Israel - June 30 - July 02, 2013*, pages 21:1–21:11, 2013.

[115] Michael Wei, John D Davis, Ted Wobber, Mahesh Balakrishnan, and Dahlia Malkhi. Beyond block i/o: implementing a distributed shared log in hardware. In *Proceedings of the 6th International Systems and Storage Conference*, page 21. ACM, 2013.

[116] Michael Wei, Amy Tai, Chris Rossbach, Ittai Abraham, Udi Wieder, Steven Swanson, and Dahlia Malkhi. Silver: A scalable, distributed, multi-versioning, always growing (ag) file system. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'16, pages 56–60, Berkeley, CA, USA, 2016. USENIX Association.

[117] Michael Wei, Amy Tai, Christopher J Rossbach, Ittai Abraham, Maithem Munshed, Medhavi Dhawan, Jim Stabile, Udi Wieder, Scott Fritchie, Steven Swanson, et al. vcorfu: A cloud-scale object store on a shared log. In *NSDI*, pages 35–49, 2017.

[118] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, 2006.

[119] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. RADOS: A Scalable, Reliable Storage Service for Petabyte-scale Storage Clusters. In *Proceedings of the 2nd International Workshop on Petascale Data Storage*, PDSW '07, 2007.

[120] Brent Welch. Exascale distributed file systems. In *MSST 2010 (Slides)*.

[121] Tom White. *Hadoop: The Definitive Guide: The Definitive Guide*, volume 1.

[122] R. Wickremesinghe, J.S. Chase, and J.S. Vitter. Distributed computing with load-managed active storage. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC 2002)*, pages 13–23, 2002.

[123] Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Darrell D. E. Long, Yangwook Kang, Zhongying Niu, and Zhipeng Tan. Design and evaluation of oasis: An active storage framework based on t10 osd standard. In *MSST '11*, Denver, CO, April 24-29 2011.

[124] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, NSDI '12.

[125] Tian Zhang and Ryan Stutsman. Javascript for extending low-latency in-memory key-value stores. In *9th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 17)*, 2017.