Storage System Design for Fast Nonvolatile Memories

by

Jacob Taylor Wires

M.Sc., Computer Science, The University of British Columbia, 2006

B.Sc., Computer Engineering, The University of California at Santa Barbara, 2004

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL
STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

October 2017

Abstract

Nonvolatile memories are transforming the data center. Over the past decade, enterprise flash has evolved to provide a thousand times more random-access throughput than mechanical disks, with a thousand times lower latency and ten times more capacity. These remarkable improvements completely reshape software concerns, allowing storage systems to take a more central role in dynamic resource management, but demanding that they do so with significantly lower overheads.

This thesis presents several novel software techniques for managing high-density storage systems. In particular, it describes a probabilistic approach to workload modeling that provides guaranteed error bounds while dramatically reducing memory overheads relative to existing state-of-the-art algorithms. It also documents the design and implementation of a storage controller that leverages dynamic constraint satisfaction techniques to continually optimize data and network flow placement for performance, efficiency, and scale.

These advances are presented within a broader design framework that provides a flexible and robust platform for managing all aspects of storage resource allocation. Informed by experiences and insights gained over six years of building an enterprise scale-out storage appliance, it is based on three key ideas: light-weight *abstraction* to decouple logical resources from physical hardware, online *analysis* to capture workload requirements, and dynamic *actuation* to adjust allocations as requirements change. Together, these capabilities allow storage software to dynamically adapt to changing workload behavior and allow stored data to play a more active role in data center computing.

Lay Summary

Most data center storage systems were originally designed to manage mechanical disks, which are some of the slowest hardware components in general computing. Enterprise flash devices and other nonvolatile memories have emerged over the past decade that are so much faster than disks that existing storage software simply cannot keep up. These devices call for new design approaches that provide efficient request processing to avoid costly performance penalties while also supporting dynamic resource management to ensure high hardware utilization. This thesis describes a system architecture and several novel software techniques that together provide this efficiency and dynamism, allowing application software to fully leverage the impressive capabilities of these new devices.

Preface

Chapters 3, 4, and 5 are versions of papers published at peer-reviewed academic conferences. They have been lightly edited for formatting.

Chapter 3

A version of Chapter 3 was published at FAST, the Usenix Conference on File and Storage Technologies, in 2014 [34]. This was a joint work with several authors. As the second author, I made significant contributions in building the system, evaluating the results, and composing the manuscript.

Chapter 4

A version of Chapter 4 was published at OSDI, the Usenix Conference on Operating Systems Design and Implementation, also in 2014 [122]. As the primary author, I set the research agenda, contributed to the implementation, evaluated the results, and presented the work. My coauthors contributed to the implementation and manuscript composition.

Chapter 5

A version of Chapter 5 was published at FAST in 2017 [120]. I was the primary author and researcher, responsible for all aspects of implementation and evaluation.

Table of Contents

List of Tables

List of Figures

List of Listings

Glossary

API application programming interface

CPU central processing unit

CRUD create, read, update, delete

DIMM dual inline memory module

FCoE fibre channel over ethernet

FIO flexible IO tester

HLL hyperloglog cardinality sketch

IO input/output

IOPS IO operations per second

IP internet protocol

IPMI intelligent platform management interface

i**SCSI** internet SCSI

JVM java virtual machine

LRU least-recently used cache replacement algorithm

LSN log serial number

M.2 specification for internally mounted expansion cards and connectors

MRC miss ratio curve

NAD network attached disk

NAND negative-and logic gate

NFS network file system protocol

NIC network interface controller

NUMA non-uniform memory access

NVDIMM nonvolatile DIMM

NVMe nonvolatile memory express

OSD object storage device

PCIe peripheral component interconnect express

PCM phase change memory

p**NFS** parallel network file system protocol

RAM random access memory

RPC remote procedure call

RTT round-trip time

SAN storage area network

SAS serial attached SCSI

SATA serial AT attachment

SCSI small computer systems interface

SDN software-defined networking

SLA service level agreement

SMART self-monitoring, analysis, and reporting technology

SMB server message block, aka common internet file system

SSD solid state drive

TCP transmission control protocol

VLAN virtual local area network

VM virtual machine

VMDK virtual machine disk

VMM virtual machine monitor

Acknowledgments

None of this work would have been possible without the help of my advisor, Andrew Warfield, who has taught me a great many things about technology, business, and life. His generosity, optimism, and enthusiasm are a wonderful inspiration, and I look forward to many more years of collaboration together.

I owe many thanks to my committee members Norm Hutchinson and Bill Aiello, who offered invaluable advice and support throughout my studies.

Finally, I would like to thank Stephen Ingram, Nick Harvey, Daniel Stodden, Kalan Macrow, Mihir Nanavati, and all my past and present colleagues at Coho Data. The laughter and discoveries shared over the years working with this team have made all the effort worthwhile.

Dedication

To my family, for all their love and support.

Chapter 1

Introduction

This thesis documents more than five years of experience building an enterprise storage appliance. The period it describes was one of remarkable hardware innovation, in which nonvolatile memories improved on the performance of mechanical disks by more than three orders of magnitude. It is a rare and exciting thing for computer scientists to witness such a dramatic transformation in such a short time. In this case, however, the advances came with a new set of problems, as existing software techniques proved ill-equipped to fully leverage new hardware capabilities. This work details some of the key challenges we faced in building a storage system designed specifically for fast, nonvolatile memories. In particular, it describes a system architecture suitable for new low-latency devices, and it presents several novel software techniques for obtaining high utility from these devices. It additionally presents a model for system design that is broadly applicable to many services within the data center. This model can be summarized by three key capabilities: light-weight *abstraction* of hardware to provide programmatic control of resource allocation, online *analysis* of workload behavior to provide insight into performance requirements, and dynamic *actuation* to adjust resource allocations as requirements change. Together, these capabilities yield robust, flexible systems. The chapters that follow, which are edited versions of published conference papers, detail how these capabilities are implemented in our production storage system.

For much of the history of computing, persistent storage has been provided by mechanical devices. From index cards to magnetic tape and finally rotating platters, the operating constraints of persistent media are limited by the physics of motion in a way that transistor-based technologies like RAM and CPUs are not. The difference is significant: while it takes roughly 100 nanoseconds to read a byte from main memory in a modern system, it takes nearly 10 milliseconds (i.e., about 100,000 times longer) to read the same byte from an enterprise-class disk. This vast disparity is commonly known as the IO *gap*, and it underpins many of the design assumptions in modern systems, affecting everything from the caching and prefetching strategies of operating systems to the on-disk layouts of file systems.

In the context of data center storage systems, this disparity has motivated *aggregated* designs that place many disks behind a single network-accessible controller. While this approach is suitable for spinning disks, it is a poor fit for new nonvolatile memories. Enterprise NVMe flash drives today can serve sequential read workloads at rates of 2.8 GiB per second, and they will only grow faster as hardware parallelism continues to increase. At these rates, the PCIe channel capacity provided by modern CPUs becomes a limiting factor, making single-head controllers impractical. At the same time, commodity NVMe drives exhibit access latencies as low as 20 microseconds, and NVDIMM-based alternatives like Intel's 3DXPoint reduce latencies even further. This exposes significant software inefficiencies as overheads that were once negligible compared to the millisecond response times of mechanical disks dominate overall request processing times on modern hardware.

Chapter 3 details our solution to these problems. *Strata* is a network-attached storage system that eliminates network and controller bottlenecks by presenting flash devices directly over the network and distributing controller logic across multiple compute nodes. Strata employs three key techniques to make this possible. First, it *virtualizes* storage devices, providing an idealized software interface that exposes sparse, virtual address spaces, allowing safe sharing among multiple clients. Second, it cleanly separates *addressing* from *placement*. Address resolution is delegated to clients via a lightweight library that provides direct access to virtual address spaces, while a centralized placement service controls the assignment of virtual address spaces to physical devices. This provides a decentralized, low-

overhead data path while allowing coordinated responses to load imbalances and hardware configuration changes. Finally, Strata leverages software defined networking to provide a scalable protocol presentation layer in support of legacy data center clients.

Strata provides the infrastructure required to support well-balanced deployments of storage, network, and compute, which is key to preventing any one of these components from becoming a bottleneck. It also augments native hardware interfaces with just enough software functionality to support safe multiplexing and dynamic resource allocation without introducing prohibitive overheads, much in the same way that CPU virtualization makes it possible to efficiently share expensive processors among multiple independent compute tasks. As a result, Strata performance scales linearly with the number of available devices. However, the decentralized architecture required to achieve this scalability adds significant engineering complexity, introducing the need for robust consensus and fencing mechanisms, among other things. Building this functionality into an enterprise storage product was a considerable undertaking that took hundreds of developer-years to complete.

By design, Strata eliminates performance bottlenecks through balanced hardware provisioning. The monetary cost of this provisioning, however, is beyond Strata's control, and the market price of NVMe devices is such that a single card can cost as much as the combined network and compute resources with which it is packaged. In other words, because the manufacturing processes of different components advance at different rates, provisioning systems with balanced performance capabilities can lead to significant *imbalances* in component costs. The best way to mitigate the effect of this imbalance on the overall cost of the system is to avoid over-provisioning expensive devices and to ensure that they achieve high utilization. This turns out to be a challenging problem for storage systems because utilization must be measured across many orthogonal dimensions, including storage and network capacity, processing power, and device queue depths.

To take just one example, storage workloads frequently exhibit access patterns that are heavily skewed towards a small proportion of their overall data sets. Serving such workloads entirely from expensive solid state devices incurs a high op-

portunity cost, because much of the devices' capabilities may be wasted storing idle data. In these cases, it may be preferable to place cold, infrequently-accessed data on cheaper, more capacious media in order to make space for hot data from other workloads. The ability to split workloads across heterogeneous devices in this manner makes it possible to allocate performance and capacity resources independently, which in turn presents opportunities for improving the utilization of high-performance media. In fact, these opportunities exist across the entire memory hierarchy, which will continue to combine devices with dramatically different performance, capacity, and cost characteristics long after mechanical disks become a thing of the past. For example, technologies like NVDIMM, 3D XPoint, and phase change memory present trade-offs in price, latency, and capacity that are almost as substantial as those between SSDs and disks.

Determining the optimal allocation for a particular workload is not trivial, however. To begin with, it is difficult to arrive at a wholly satisfying definition of 'optimal' in this context. But setting that aside for the moment, even simply identifying hot data can be computationally expensive. Given modern hardware, a single storage workload may be capable of generating hundreds of thousands of requests per second across billions of unique addresses – and it may remain active for weeks, months, or even years on end. The effort involved in analyzing such voluminous request streams can be immense. Indeed, shortly after we began investigating how we might improve flash utilization in Strata, we ran into exactly this problem: applying classical stack distance analysis techniques to a week-long storage trace of just a handful of machines required roughly an hour of compute time and 92 GiB of RAM. This was inconvenient for our research, but downright prohibitive for use in production.

Chapter 4 presents a novel locality analysis technique we developed that is computationally tractable even for very large workloads. The technique leverages probabilistic data structures called *counter stacks* to enable approximate LRU stack distance analysis with sublinear memory overheads. This represents a significant improvement over the previous state of the art, allowing us to analyze the above-mentioned week-long trace in under twenty minutes with just 80 MiB of RAM (less than a tenth of a percent of what was previously required). Counter stacks improve

on existing analysis techniques in two additional ways. First, they make it possible to analyze how workload locality changes over time, revealing phase changes and other temporal patterns that can be exploited to improve performance and efficiency. Second, they allow us to model how workloads interact with each other on shared storage. For example, we can quickly calculate the degree to which unrelated workloads would interfere with each other if placed on the same device, allowing us to make informed decisions when distributing data across storage nodes.

Counter stacks get their name from the cardinality counters they rely on to track data references over time. By combining knowledge of the unique addresses accessed over various time windows with a record of the total number of requests over the same windows, counter stacks give an indication of a workload's temporal locality. For the sake of practicality, counter stacks use approximate counters that belong to a class of *streaming* algorithms and data structures designed for processing very large data sets. Streaming algorithms often trade accuracy for efficiency and are well-suited for scenarios where imperfect results can be tolerated. For example, they enable efficient estimation of join sizes when optimizing database queries, and they support anomaly detection of traffic patterns in large networks. We suspect that with a bit of creativity, they will prove useful for analyzing and tuning many other aspects of system performance in unforeseen ways; indeed, cardinality sketches form the basis of another data structure we recently developed for measuring implicit sharing among copy-on-write snapshots. Investigating further opportunities for integrating streaming algorithms into high-throughput, performance-sensitive systems is a promising direction for future research.

Counter stacks enable a degree of continuous workload analysis that was previously impractical. The visibility they provide into capacity and performance requirements help to inform decisions about how to distribute data across heterogeneous devices. However, these are just two of a large number of criteria that must be considered when allocating system resources. Other salient examples include the need to maintain hardware redundancy for replicated data and the desire to balance network load across available links. In fact, in a disaggregated, heterogeneous system like Strata, deciding how best to accommodate all of these objectives is in

itself a challenging problem, especially since, in some circumstances, one objective may directly contradict another. This is an important problem, however, because poor decisions can have calamitous effects on system performance, reliability, and efficiency.

Chapter 5 describes *Mirador*, a dynamic resource management service designed for network-attached storage systems. Informed by the detailed profiling and analysis made possible by counter stacks and leveraging the device and protocol virtualization provided by Strata, Mirador strives to achieve high hardware utilization and system availability by dynamically migrating workloads in response to changing requirements. In many ways, it can be likened to the centralized controllers of traditional aggregated storage systems: it maintains a global view of resource utilization and workload behavior and it provides a unified control path for managing resource allocation. It is more sophisticated than typical controllers, however, because its purview includes the entire back-end storage network: it controls how client connections are routed to storage servers as well as how data is placed on available devices. Moreover, it takes an unconventionally proactive approach to resource management, continually seeking adjustments that might improve performance and utility. This approach is made feasible by the high random-access bandwidth of solid state devices, which dramatically lowers the performance cost of migrating data relative to spinning disks.

Managing the many moving parts of a large system like Strata is complicated. Even simply formulating an allocation policy that is suitable for all possible contingencies is a challenging task. The configuration space is large and multidimensional, and attempting to anticipate every potential corner case is time-consuming and error-prone. Mirador addresses this complexity by providing a framework for codifying policies as a collection of simple, independent *objective functions*, each of which describes an allocation strategy for a single resource. Objective functions are assigned numerical *costs* that induce a priority ordering for situations where not all goals can be met. Mirador combines these objective functions with established optimization techniques to efficiently *search* the configuration space for preferable alternatives while maintaining the invariants necessary to guarantee resiliency. This approach has a number of appealing properties. It allows domain ex-

perts to define specific allocation goals without prescribing how the system should behave as a whole. It naturally supports incremental updates to allocation policies so that new classes of workloads and hardware can be more easily accommodated. And it makes the system more robust to workload hot spots and hardware faults by facilitating continuous, dynamic optimization.

Strata's design eliminates network and controller hardware as performance bottlenecks, but it cannot eliminate the more general problem of resource scarcity. For example, as deployments expand across racks, top-of-rack switching becomes a limited resource that must be allocated frugally. Mirador addresses this particular problem by leveraging its knowledge of the relative availability of local and remote bandwidth (as codified by policy objective functions) to coordinate the migration of data and client connections in order to avoid cross-rack traffic. But more important than the specific balance that Strata and Mirador strike with the current generation of hardware is the support they provide for dynamically responding to resource scarcity and workload imbalances in general.

Figure 1.1 presents a schematic overview of how the three components described in this thesis work together to achieve this level of dynamism. By providing carefully-considered software abstractions – both in virtualizing hardware to decouple logical resources from the underlying physical devices, and in cleanly separating control- and data-path logic – Strata provides flexible, programmatic control of storage and network resources. By enabling efficient, accurate working set analysis techniques, counter stacks provide insight into the performance and capacity requirements of client workloads in live systems. And by leveraging these capabilities to actuate system-level responses to shifting resource consumption, Mirador is able to continuously optimize for performance, efficiency, and reliability.

Indeed, the central claim of this thesis is that data center storage systems – and most data center services in general – should be carefully designed to enable flexible, robust, and dynamic responses to changes in both workload behavior and hardware capabilities. In a large, multi-tenant environment like a data center, diverse and varying workload behavior is inevitable. And as evidenced by the revolutionary advances of storage devices over the past few years, even long-standing

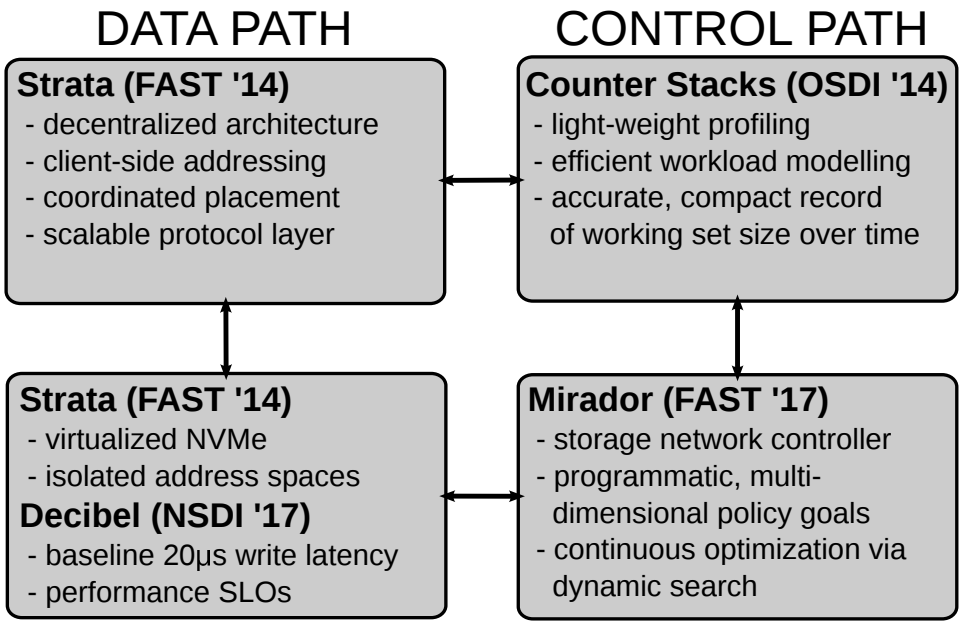


Figure 1.1: Schematic overview of system contributions (see § 1.1 and Chapter 6 for more details about Decibel)

assumptions about the relative performance of hardware components must be reconsidered from time to time. Consequently, robust systems should be capable of automatically adapting to changes across all of these dimensions, both at the scale of scheduling epochs and hardware life cycles. We demonstrate in the following chapters how abstraction, analysis, and actuation can be combined to provide this responsiveness in a decentralized storage system with exacting performance requirements. More generally, we believe that these capabilities provide a sound framework for a broad class of data center services as workloads and hardware continue to evolve.

1.1 Publications

The work presented in this thesis is based on a selection of three closely-related publications. Below I present the complete list of research papers to which I contributed over the course of my studies.

Strata is our scale-out storage architecture for fast nonvolatile memories [34].

Mirador codifies allocation policies as individual objective functions and uses established constraint satisfaction techniques to continually optimize the placement of data and network flows in Strata [120].

Decibel extends the work of Strata to present a new volume abstraction that manages compute and network resources to provide contention-free request processing for ultra low-latency devices [85].

Counter Stacks are a novel probabilistic data structure for recording working sets over time. They enable the calculation of miss ratio curves with sublinear memory overheads, a dramatic improvement over the previous state of the art [122].

Approximating Hit Rate Curves Using Streaming Algorithms is a companion paper that presents a thorough analysis of the accuracy and computational complexity of counter stacks [41].

Ownership Sketches are a novel data structure, inspired by counter stacks, that enable efficient tracking of implicit sharing in copy on write snapshots [123].

MapFS explores the possibility of exposing file system address space mappings to userspace by providing efficient splice operations on file data [121]. This work helped motivate the flexible addressing schemes provided by Strata.

Capo demonstrates how local disks can be used as client-side caches to reduce load on shared storage servers [101]. It also stands as an early example of the data-driven design approach that ultimately led to counter stacks.

Dovetail presents a framework for safely upgrading on-disk data structures in cloud storage platforms while minimizing the impact on client workloads [82]. Its support for non-disruptive system reconfiguration was a precursor to the dynamic resource management provided by Mirador.

Block Mason is a virtual block device framework that supports modular, stackable userspace implementations for enhanced flexibility and portability [81]. Its composable data path served as a model for Strata's request dispatching architecture.

Chapter 2

Nonvolatile Memory

The reign of spinning disks as the predominant technology in enterprise storage is coming to an end. While hard drives have stagnated because of physical limitations on rotational speed, nonvolatile technologies like NAND flash and phase change memory (PCM) have flourished, bridging the gap between RAM and disk. NAND flash, long common in cameras and mobile phones, has recently become a viable alternative to magnetic disks thanks to dramatic improvements in performance, reliability, and affordability: enterprise flash devices today provide random-access throughput that is a thousand times greater than mechanical disks at latencies that are a thousand times lower, while remaining cost-competitive with their rotating counterparts. Emerging technologies like phase change and spin-transfer torque memories avoid transistor scaling difficulties by using entirely different physical-chemical mechanisms to provide bit storage, promising additional performance and endurance improvements. The impact of these innovations can be broadly categorized according to three trends, each of which is changing the data center in important ways. First, increased performance density has effectively inverted the IO gap, violating many of the assumptions behind conventional storage designs. Second, increased capacity density is placing new demands on device connectivity and raising serious concerns about failure recovery times. And third, reduced power and space requirements are altering the physical and logistical constraints

of hardware deployment. These advances solve many long-standing problems in storage design, but they also present new challenges.

Perhaps the most remarkable characteristic of nonvolatile memories is their radical performance density. The difference relative to magnetic disks in this regard is really one of kind rather than degree. By eliminating the mechanical component of storage hardware, solid state technologies remove the single largest contributor to the IO gap. This reduces access times in absolute terms, but, more importantly, it does away with the armature movement and rotational latency that together impose additive, millisecond-granularity penalties under random-access workloads. This makes spatial locality much less important than it used to be, allowing nonvolatile memory to be virtualized without significant performance degradation. Indeed, flash firmware does just this in the translation layers that manage erase cycles and provide wear levelling, as does storage software in providing deduplication to increase effective capacity. High random-access throughput also makes it feasible to migrate data to balance load and improve efficiency without affecting primary workloads, allowing stored data to play a much more active role in computation.

This increase in performance density has been enabled in part by the migration of storage devices onto wider, faster interfaces as nonvolatile memories have moved from SAS and SATA buses to PCIe and DIMM alternatives that provide higher throughput at much lower latency. In fact, the latency of modern PCIe flash devices is so low that avoiding processing overhead has become a significant challenge. A similar trend has emerged in networking software as commodity Ethernet transmission rates have increased by a thousandfold, from 100 megabits per second in 1995 to 100 gigabits per second today. But the transformation has been more profound for storage systems, which have become a million times faster over the same period. This has important consequences for storage software, which can no longer assume that processing is effectively free but must instead contend with nanosecond request deadlines.

In addition to offering lower latency, the increased parallelism of PCIe devices makes it easier to share hardware among many workloads. The NVMe specification allows up to 65,000 queues per device, and while few vendors actually provide

anywhere near this many, most offer enough to make it possible to completely partition data structures, interrupt handling, and request processing across cores without any need for thread synchronization. Exposing this parallelism directly to software layers inverts the traditional model of a single elevator scheduler mediating access to multiple spindles, but it is crucial for achieving full device saturation.

The increasing capacity density of nonvolatile memories, while perhaps less spectacular than the coincident performance improvements, is also changing the data center in important ways. Although magnetic disk capacity has plateaued at about 10 TB, with shingled magnetic recording offering further marginal improvements suitable primarily for read-mostly workloads, nonvolatile memory capacity is increasing at a steady pace. Transistor-based media like NAND flash continue to benefit from die process improvements that lead to smaller, denser memories at a rate roughly in line with Moore's Law, even as processors have already pushed these gains close to the limit. Additional innovations like multi-level cells on NAND flash and the three-dimensional circuit layouts of 3D XPoint lead to even further capacity gains, although often at the expense of increased bit error rates and reduced write performance. Thanks to these advances, the single-device capacity of nonvolatile memories will soon surpass that of spinning disks by more than an order of magnitude.

This capacity density poses new challenges, however. Modern enterprise CPU microarchitectures like Intel's Skylake series provide around 40 PCIe lanes per controller hub. In a typical network-facing server, roughly half of these might be dedicated to NICs, leaving only twenty lanes to connect to storage hardware. This means that PCIe switching is needed to host even a moderate number of NVMe devices (which consume four lanes each) in a single machine. Under these constraints, PCIe throughput quickly becomes an issue as device capacity scales. Given their compact size, it is not unreasonable to imagine hosting upwards of 32 flash cards in a single 1U server; at 128 TB per device, this would mean exposing 4 PB of storage over PCIe lanes with a combined throughput of just 200 GB/sec, severely limiting overall data access rates. This problem extends beyond individual hosts as well: links between top-of-rack switches are typically oversubscribed at a ratio of three or four to one, presenting another potential bottleneck. Capacious devices

also present new difficulties in maintaining data resiliency. Repairing redundancy when multi-terabyte devices fail can be time-consuming, increasing exposure to permanent data loss. These factors put new demands on storage software, which must place data carefully to mitigate transport bottlenecks and arrange for fast recovery times.

These performance and capacity gains come with significant *reductions* in power consumption and physical size. Because nonvolatile memories contain no moving parts, they consume only around half a watt when idle, and twice that under load. Rotating disks, on the other hand, consume around 5 watts per spindle when idle, and twice that again under load. Furthermore, opportunistic efforts to reduce power consumption by powering down idle disks are generally impractical without advanced knowledge of workload patterns, because disks take seconds to spin back up. Combined with the fact that large arrays of spindles are typically required to achieve even moderate random-access performance, this places tremendous burdens on data center power and cooling systems. At the same time, nonvolatile memories can be packaged much more compactly than spinning platters. For example, thanks to new form factors like M.2, it is reasonable to imagine a single 2U enclosure providing adequate flash storage for an entire rack of machines, replacing many hundreds of disks in so doing. Along with the performance and capacity density described above, this physical compactness makes disaggregated architectures a much more natural fit for nonvolatile memories than traditional SAN designs.

Nonvolatile memories provide orders of magnitude more performance and capacity than mechanical disks while requiring substantially less power and physical space. In so doing, they completely reshape storage software concerns. Rather than batching requests to avoid seeks, software must restrict processing times to microseconds or less. Rather than aggregating disk arrays to increase parallelism, systems need to expose individual device queues with minimal cross-core synchronization. Rather than uniformly distributing data across spindles, controllers should consider migrating data in response to load imbalances and hot spots. In short, storage software needs to become significantly more efficient, flexible, and dynamic if it is to fully realize the potential of these exciting new hardware technologies.

Chapter 3

Strata: Scalable High-Performance Storage on Virtualized Non-Volatile Memory

A version of this chapter was published at the 12th USENIX Conference on File and Storage Technologies in 2014 [34].

3.1 Introduction

Flash-based storage devices are fast, expensive, and demanding: a single device is capable of saturating a 10Gb/s network link (even for random IO), consuming significant CPU resources in the process. That same device may cost as much as (or more than) the server in which it is installed[1]. The cost and performance characteristics of fast, non-volatile media have changed the calculus of storage system design and present new challenges for building efficient and high-performance datacenter storage.

[1]Enterprise-class PCIe flash drives in the 1TB capacity range currently carry list prices in the range of $3-5K USD. Large-capacity, high-performance cards are available for list prices of up to $160K.

This chapter describes the architecture of a commercial flash-based network-attached storage system, built using commodity hardware. In designing the system around PCIe flash, we begin with two observations about the effects of high-performance drives on large-scale storage systems. First, these devices are fast enough that in most environments, many concurrent workloads are needed to fully saturate them, and even a small degree of processing overhead will prevent full utilization. Thus, we must change our approach to the media from *aggregation* to *virtualization*. Second, aggregation is still necessary to achieve properties such as redundancy and scale. However, it must avoid the performance bottleneck that would result from the monolithic controller approach of a traditional storage array, which is designed around the obsolete assumption that media is the slowest component in the system. Further, to be practical in existing datacenter environments, we must remain compatible with existing client-side storage interfaces and support standard enterprise features like snapshots and deduplication.

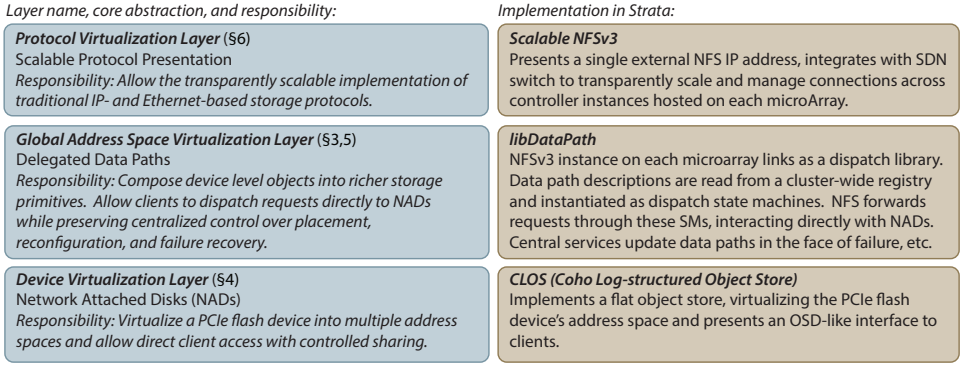


Figure 3.1: Strata network storage architecture

In this chapter we explore the implications of these two observations on the design of a scalable, high-performance NFSv3 implementation for the storage of virtual machine images. Our system is based on the building blocks of PCIe flash in commodity x86 servers connected by 10 gigabit switched Ethernet. We describe two broad technical contributions that form the basis of our design:

1. A delegated mapping and request dispatch interface from client data to physical resources through *global data address virtualization*, which allows clients

to directly address data while still providing the coordination required for online data movement (e.g., in response to failures or for load balancing).

2. SDN-assisted *storage protocol virtualization* that allows clients to address a single virtual protocol gateway (e.g., NFS server) that is transparently scaled out across multiple real servers. We have built a scalable NFS server using this technique, but it applies to other protocols (such as iSCSI, SMB, and FCoE) as well.

At its core, Strata uses device-level object storage and dynamic, global address-space virtualization to achieve a clean and efficient separation between control and data paths in the storage system. Flash devices are split into virtual address spaces using an object storage-style interface, and clients are then allowed to directly communicate with these address spaces in a safe, low-overhead manner. In order to compose richer storage abstractions, a global address space virtualization layer allows clients to aggregate multiple per-device address spaces with mappings that achieve properties such as striping and replication. These delegated address space mappings are coordinated in a way that preserves direct client communications with storage devices, while still allowing dynamic and centralized control over data placement, migration, scale, and failure response.

Serving this storage over traditional protocols like NFS imposes a second scalability problem: clients of these protocols typically expect a single server IP address, which must be dynamically balanced over multiple servers to avoid being a performance bottleneck. In order to both scale request processing and to take advantage of full switch bandwidth between clients and storage resources, we developed a *scalable protocol presentation layer* that acts as a client to the lower layers of our architecture, and that interacts with a software-defined network switch to scale the implementation of the protocol component of a storage controller across arbitrarily many physical servers. By building protocol gateways as clients of the address virtualization layer, we preserve the ability to delegate scale-out access to device storage without requiring interface changes on the end hosts that consume the storage.

3.2 Architecture

The performance characteristics of emerging storage hardware demand that we completely reconsider storage architecture in order to build scalable, low-latency shared persistent memory. The reality of deployed applications is that interfaces must stay exactly the same in order for a storage system to have relevance. Strata's architecture aims to take a step toward the first of these goals, while keeping a pragmatic focus on the second.

Figure 3.1 characterizes the three layers of Strata's architecture. The goals and abstractions of each layer of the system are on the left-hand column, and the concrete embodiment of these goals in our implementation is on the right. At the base, we make devices accessible over an object storage interface, which is responsible for virtualizing the device's address space and allowing clients to interact with individual virtual devices. This approach reflects our view that system design for these storage devices today is similar to that of CPU virtualization ten years ago: devices provide greater performance than is required by most individual workloads and so require a lightweight interface for controlled sharing in order to allow multi-tenancy. We implement a per-device object store that allows a device to be virtualized into an address space of 2^{128} sparse objects, each of which may be up to 2^{64} bytes in size. Our implementation is similar in intention to the OSD specification, itself motivated by network attached secure disks [50]. While not broadly deployed to date, device-level object storage is receiving renewed attention today through pNFS's use of OSD as a backend, the NVMe *namespace* abstraction, and in emerging hardware such as Seagate's Kinetic drives [99]. Our object storage interface as a whole is not a significant technical contribution, but it does have some notable interface customizations described in § 3.4. We refer to this layer as a *Network Attached Disk*, or NAD.

The middle layer of our architecture provides a global address space that supports the efficient composition of *IO processor*s that translate client requests on a *virtual object* into operations on a set of NAD-level *physical objects*. We refer to the graph of IO processors for a particular virtual object as its *data path*, and we maintain the description of the data path for every object in a global *virtual address*

map. Clients use a dispatch library to instantiate the processing graph described by each data path and perform direct IO on the physical objects at the leaves of the graph. The virtual address map is accessed through a coherence protocol that allows central services to update the data paths for virtual objects while they are in active use by clients. More concretely, data paths allow physical objects to be composed into richer storage primitives, providing properties such as striping and replication. The goal of this layer is to strike a balance between scalability and efficiency: it supports direct client access to device-level objects, without sacrificing central management of data placement, failure recovery, and more advanced storage features such as deduplication and snapshots.

Finally, the top layer performs *protocol virtualization* to allow clients to access storage over standard protocols (such as NFS) without losing the scalability of direct requests from clients to NADs. The presentation layer is tightly integrated with a 10Gb software-defined Ethernet switching fabric, allowing external clients the illusion of connecting to a single TCP endpoint, while transparently and dynamically balancing traffic to that single IP address across protocol instances on all of the NADs. Each protocol instance is a thin client of the layer below, which may communicate with other protocol instances to perform any additional synchronization required by the protocol (e.g., to maintain NFS namespace consistency).

The mapping of these layers onto the hardware that our system uses is shown in Figure 3.2. Requests travel from clients into Strata through an OpenFlow-enabled switch, which dispatches them according to load to the appropriate protocol handler running on a *MicroArray (µArray)* — a small host configured with flash devices and enough network and CPU to saturate them, containing the software stack representing a single NAD. For performance, each of the layers is implemented as a library, allowing a single process to handle the flow of requests from client to media. The NFSv3 implementation acts as a client of the underlying dispatch layer, which transforms requests on virtual objects into one or more requests on physical objects, issued through function calls to local physical objects and by RPC to remote objects. While the focus of the rest of this chapter is on this concrete implementation of scale-out NFS, it is worth noting that the design is intended to allow applications the opportunity to link directly against the same data path library that

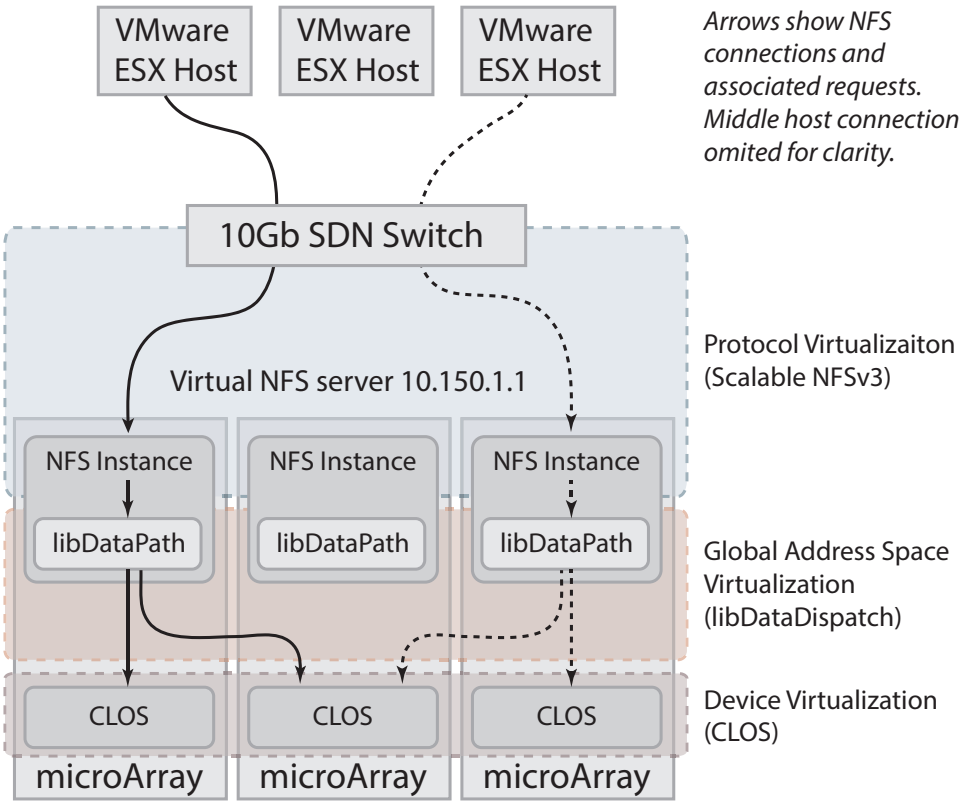


Figure 3.2: Hardware view of a Strata deployment

the NFS implementation uses, resulting in a multi-tenant, multi-presentation storage system with a minimum of network and device-level overhead.

3.2.1 Scope of this Work

There are three aspects of our design that are not considered in detail within this presentation. First, we only discuss NFS as a concrete implementation of protocol virtualization. Strata has been designed to host and support multiple protocols and tenants, but our initial product release is specifically NFSv3 for VMware clients, so we focus on this type of deployment in describing the implementation. Second, Strata was initially designed to be a software layer that is co-located on the same physical servers that host virtual machines. We have moved to a separate physical

hosting model where we directly build on dedicated hardware, but there is nothing that prevents the system from being deployed in a more co-located (or "converged") manner. Finally, our full implementation incorporates a tier of spinning disks on each of the storage nodes to allow cold data to be stored more economically behind the flash layer. However, in this chapter we configure and describe a single-tier, all-flash system to simplify the exposition.

In the next sections we discuss three relevant aspects of Strata—address space virtualization, dynamic reconfiguration, and scalable protocol support—in more detail. We then describe some specifics of how these three components interact in our NFSv3 implementation for VM image storage before providing a performance evaluation of the system as a whole.

3.3 Data Paths

Strata provides a common library interface to data that underlies the higher-level, client-specific protocols described in § 3.6. This library presents a notion of virtual objects, which are available cluster-wide and may comprise multiple physical objects bundled together for parallel data access, fault tolerance, or other reasons (e.g., data deduplication). The library provides a superset of the object storage interface provided by the NADs (§ 3.4), with additional interfaces to manage the placement of objects (and ranges within objects) across NADs, to maintain data invariants (e.g., replication levels and consistent updates) when object ranges are replicated or striped, and to coordinate both concurrent access to data and concurrent manipulation of the *virtual address maps* describing their layout.

To avoid IO bottlenecks, users of the data path interface (which may be native clients or protocol gateways such as our NFS server) access data directly. To do so, they map requests from virtual objects to physical objects using the virtual address map. This is not simply a pointer from a virtual object *(id, range)* pair to a set of physical object *(id, range)* pairs. Rather, each virtual range is associated with a particular *processor* for that range, along with processor-specific context. Strata uses a dispatch-oriented programming model in which a pipeline of operations is

performed on requests as they are passed from an originating client, through a set of transformations, and eventually to the appropriate storage device(s). Our model borrows ideas from packet processing systems such as X-Kernel [60], Scout [84], and Click [69], but adapts them to a storage context, in which modules along the pipeline perform translations through a set of layered address spaces, and may fork and/or collect requests and responses as they are passed.

The dispatch library provides a collection of request processors, which can stand alone or be combined with other processors. Each processor takes a storage request (e.g., a read or write request) as input and produces one or more requests to its children. NADs expose isolated sparse objects; processors perform translations that allow multiple objects to be combined for some functional purpose, and present them as a single object, which may in turn be used by other processors. The idea of request-based address translation to build storage features has been used in other systems [74, 75, 80], often as the basis for volume management; Strata disentangles it from the underlying storage system and treats it as a first-class dispatch abstraction.

The composition of dispatch modules bears similarity to Click [69], but the application in a storage domain carries a number of differences. First, requests are generally acknowledged at the point that they reach a storage device, and so as a result they differ from packet forwarding logic in that they travel both down and then back up through a dispatch stack; processors contain logic to handle both requests *and* responses. Second, it is common for requests to be split or merged as they traverse a processor — for example, a replication processor may duplicate a request and issue it to multiple nodes, and then collect all responses before passing a single response back up to its parent. Finally, while processors describe fast, library-based request dispatching logic, they typically depend on additional facilities from the system. Strata allows processor implementations access to APIs for shared, cluster-wide state which may be used on a control path to, for instance, store replica configuration. It additionally provides facilities for background functionality such as NAD failure detection and response. The intention of the processor organization is to allow dispatch decisions to be pushed out to client implementations and be made with minimal performance impact, while still benefiting from

common system-wide infrastructure for maintaining the system and responding to failures. The responsibilities of the dispatch library are described in more detail in the following subsections.

3.3.1 The Virtual Address Map

```
/objects/112:
    type=regular dispatch={object=111
                           type=dispatch}

/objects/111:
    type=dispatch
    stripe={stripecount=8 chunksize=524288
        0={object=103 type=dispatch}
        1={object=104 type=dispatch}}

/objects/103:
    type=dispatch
    rpl={policy=mirror storecount=2
            {storeid=a98f2... state=in-sync}
            {storeid=fc89f... state=in-sync}}
```

Figure 3.3: Virtual object to physical object range mapping

Figure 3.3 shows the relevant information stored in the virtual address map for a typical object. Each object has an identifier, a type, some type-specific context, and may contain other metadata such as cached size or modification time information (which is not canonical, for reasons discussed below).

The entry point into the virtual address map is a *regular* object. This contains no location information on its own, but delegates to a top-level *dispatch* object. In Figure 3.3, object 112 is a regular object that delegates to a dispatch processor whose context is identified by object 111 (the IDs are in reverse order here because the dispatch graph is created from the bottom up, but traversed from the top down). Thus when a client opens file 112, it instantiates a dispatcher using the data in object 111 as context. This context informs the dispatcher that it will be delegating IO through a *striped* processor, using 2 stripes for the object and a stripe width of 512K. The dispatcher in turn instantiates 8 processors (one for each stripe), each configured with the information stored in the object associated with each stripe

(e.g., stripe 0 uses object 103). Finally, when the stripe dispatcher performs IO on stripe 0, it will use the context in the object descriptor for object 103 to instantiate a *replicated* processor, which mirrors writes to the NADs listed in its replica set, and issues reads to the nearest in sync replica (where distance is currently simply local or remote).

In addition to the striping and mirroring processors described here, the map can support other more advanced processors, such as erasure coding, or byte-range mappings to arbitrary objects (which supports among other things data deduplication).

3.3.2 Dispatch

IO requests are handled by a chain of dispatchers, each of which has some common functionality. Dispatchers may have to fragment requests into pieces if they span the ranges covered by different subprocessors, or clone requests into multiple subrequests (e.g., for replication), and they must collect the results of subrequests and deal with partial failures.

The replication and striping modules included in the standard library are representative of the ways processors transform requests as they traverse a dispatch stack. The replication processor allows a request to be split and issued concurrently to a set of replica objects. The request address remains unchanged within each object, and responses are not returned until all replicas have acknowledged a request as complete. The processor prioritizes reading from local replicas, but forwards requests to remote replicas in the event of a failure (either an error response or a timeout). It imposes a global ordering on write requests and streams them to all replicas in parallel. It also periodically commits a light-weight checkpoint to each replica's log to maintain a persistent record of synchronization points; these checkpoints are used for crash recovery (§ 3.5.1).

The striping processor distributes data across a collection of sparse objects. It is parameterized to take a stripe size (in bytes) and a list of objects to act as the ordered stripe set. In the event that a request crosses a stripe boundary, the processor splits

that request into a set of per-stripe requests and issues those asynchronously, collecting the responses before returning. Static, address-based striping is a relatively simple load balancing and data distribution mechanism as compared to placement schemes such as consistent hashing [64]. Our experience has been that the approach is effective, because data placement tends to be reasonably uniform within an object address space, and because using a reasonably large stripe size (we default to 512KB) preserves locality well enough to keep request fragmentation overhead low in normal operation.

3.3.3 Coherence

Strata clients also participate in a simple coordination protocol in order to allow the virtual address map for a virtual object to be updated even while that object is in use. Online reconfiguration provides a means for recovering from failures, responding to capacity changes, and even moving objects in response to observed or predicted load (on a device basis — this is distinct from client load balancing, which we also support through a switch-based protocol described in § 3.6.2).

The virtual address maps are stored in a distributed, synchronized *configuration database* implemented over Apache Zookeeper, which is also available for any low-bandwidth synchronization required by services elsewhere in the software stack. The coherence protocol is built on top of the configuration database. It is currently optimized for a single writer per object, and works as follows: when a client wishes to write to a virtual object, it first claims a lock for it in the configuration database. If the object is already locked, the client requests that the holder release it so that the client can claim it. If the holder does not voluntarily release it within a reasonable time, the holder is considered unresponsive and fenced from the system using the mechanism described in § 3.6.2. This is enough to allow movement of objects, by first creating new, out of sync physical objects at the desired location, then requesting a release of the object's lock holder if there is one. The user of the object will reacquire the lock on the next write, and in the process discover the new out of sync replica and initiate resynchronization. When the new replica is in sync, the same process may be repeated to delete replicas that are at

undesirable locations.

3.4 Network Attached Disks

The unit of storage in Strata is a Network Attached Disk (NAD), consisting of a balanced combination of CPU, network and storage components. In our current hardware, each NAD has two 10 gigabit Ethernet ports, two PCIe flash cards capable of 10 gigabits of throughput each, and a pair of Xeon processors that can keep up with request load and host additional services alongside the data path. Each NAD provides two distinct services. First, it efficiently multiplexes the raw storage hardware across multiple concurrent users, using an object storage protocol. Second, it hosts applications that provide higher level services over the cluster. Object rebalancing (§ 3.5.2) and the NFS protocol interface (§ 3.6.1) are examples of these services.

At the device level, we multiplex the underlying storage into objects, named by 128-bit identifiers and consisting of sparse 2^{64} byte data address spaces. These address spaces are currently backed by a garbage-collected log-structured object store, but the implementation of the object store is opaque to the layers above and could be replaced if newer storage technologies made different access patterns more efficient. We also provide increased capacity by allowing each object to flush low priority or infrequently used data to disk, but this is again hidden behind the object interface. The details of disk tiering, garbage collection, and the layout of the file system are beyond the scope of this chapter.

The physical object interface is for the most part a traditional object-based storage device [98, 99] with a CRUD interface for sparse objects, as well as a few extensions to assist with our clustering protocol (§ 3.5.1). It is significantly simpler than existing block device interfaces, such as the SCSI command set, but is also intended to be more direct and general purpose than even narrower interfaces such as those of a key-value store. Providing a low-level hardware abstraction layer allows the implementation to be customized to accommodate best practices of individual flash implementations, and also allows more dramatic design changes at the media

interface level as new technologies become available.

3.4.1 Network Integration

As with any distributed system, we must deal with misbehaving nodes. We address this problem by tightly coupling with managed Ethernet switches, which we discuss at more length in § 3.6.2. This approach borrows ideas from systems such as Sane [26] and Ethane [27], in which a managed network is used to enforce isolation between independent endpoints. The system integrates with both OpenFlow-based switches and software switching at the VMM to ensure that Strata objects are only addressable by their authorized clients.

Our initial implementation used Ethernet VLANs, because this form of hardware-supported isolation is in common use in enterprise environments. In the current implementation, we have moved to OpenFlow, which provides a more flexible tunneling abstraction for traffic isolation.

We also expose an isolated private virtual network for out-of-band control and management operations internal to the cluster. This allows NADs themselves to access remote objects for peer-wise resynchronization and reorganization under the control of a cluster monitor.

3.5 Online Reconfiguration

There are two broad categories of events to which Strata must respond in order to maintain its performance and reliability properties. The first category includes faults that occur directly on the data path. The dispatch library recovers from such faults immediately and automatically by reconfiguring the affected virtual objects on behalf of the client. The second category includes events such as device failures and load imbalance. These are handled by a dedicated *cluster monitor* which performs large-scale reconfiguration tasks to maintain the health of the system as a whole. In all cases, reconfiguration is performed online and has minimal impact on client availability.

3.5.1 Object Reconfiguration

A number of error recovery mechanisms are built directly into the dispatch library. These mechanisms allow clients to quickly recover from failures by reconfiguring individual virtual objects on the data path.

IO Errors

The replication IO processor responds to read errors in the obvious way: by immediately resubmitting failed requests to different replicas. In addition, clients maintain per-device error counts; if the aggregated error count for a device exceeds a configurable threshold, a background task takes the device offline and coordinates a system-wide reconfiguration (§ 3.5.2).

IO processors respond to write errors by synchronously reconfiguring virtual objects at the time of the failure. This involves three steps. First, the affected replica is marked *out of sync* in the configuration database. This serves as a global, persistent indication that the replica may not be used to serve reads because it contains potentially stale data. Second, a best-effort attempt is made to inform the NAD of the error so that it can initiate a background task to resynchronize the affected replica. This allows the system to recover from transient failures almost immediately. Finally, the IO processor allocates a special *patch* object on a separate device and adds this to the replica set. Once a replica has been marked out of sync, no further writes are issued to it until it has been resynchronized; patches prevent device failures from impeding progress by providing a temporary buffer to absorb writes under these degraded conditions. With the patch object allocated, the IO processor can continue to meet the replication requirements for new writes while out of sync replicas are repaired in the background. A replica set remains available as long as an in sync replica or an out of sync replica *and* all of its patches are available.

Resynchronization

In addition to providing clients direct access to devices via virtual address maps, Strata provides a number of background services to maintain the health of individual virtual objects and the system as a whole. The most fundamental of these is the *resync* service, which provides a background task that can resynchronize objects replicated across multiple devices.

Resync is built on top of a special NAD `resync` API that exposes the underlying log structure of the object stores. NADs maintain a Log Serial Number (LSN) with every physical object in their stores; when a record is appended to an object's log, its LSN is monotonically incremented. The IO processor uses these LSNs to impose a global ordering on the changes made to physical objects that are replicated across stores and to verify that all replicas have received all updates.

If a write failure causes a replica to go out of sync, the client can request the system to resynchronize the replica. It does this by invoking the `resync` RPC on the NAD which hosts the out of sync replica. The server then starts a background task which streams the missing log records from an in sync replica and applies them to the local out of sync copy, using the LSN to identify which records the local copy is missing.

During resync, the background task has exclusive write access to the out of sync replica because all clients have been reconfigured to use patches. Thus the resync task can chase the tail of the in sync object's log while clients continue to write. When the bulk of the data has been copied, the resync task enters a final *stop-and-copy* phase in which it acquires exclusive write access to all replicas in the replica set, finalizes the resync, applies any client writes received in the interim, marks the replica as in sync in the configuration database, and removes the patch.

It is important to ensure that resync makes timely progress to limit vulnerability to data loss. Very heavy client write loads may interfere with resync tasks and, in the worst case, result in unbounded transfer times. For this reason, when an object is under resync, client writes are throttled and resync requests are prioritized.

Crash Recovery

Special care must be taken in the event of an unclean shutdown. On a clean shutdown, all objects are released by removing their locks from the configuration database. Crashes are detected when replica sets are discovered with stale locks (i.e., locks identifying unresponsive IO processors). When this happens, it is not safe to assume that replicas marked *in sync* in the configuration database are truly in sync, because a crash might have occured midway through a the configuration database update; instead, all the replicas in the set must be queried directly to determine their states.

In the common case, the IO processor retrieves the LSN for every replica in the set and determines which replicas, if any, are out of sync. If all replicas have the same LSN, then no resynchronization is required. If different LSNs are discovered, then the replica with the highest LSN is designated as the authoritative copy, and all other replicas are marked out of sync and resync tasks are initiated.

If a replica cannot be queried during the recovery procedure, it is marked as *diverged* in the configuration database and the replica with the highest LSN from the remaining available replicas is chosen as the authoritative copy. In this case, writes may have been committed to the diverged replica that were not committed to any others. If the diverged replica becomes available again some time in the future, these extra writes must be discarded. This is achieved by rolling the replica back to its last checkpoint and starting a resync from that point in its log. Consistency in the face of such rollbacks is guaranteed by ensuring that objects are successfully marked out of sync in the configuration database *before* writes are acknowledged to clients. Thus write failures are guaranteed to either mark replicas out of sync in the configuration database (and create corresponding patches) or propagate back to the client.

3.5.2 System Reconfiguration

Strata also provides a highly-available monitoring service that watches over the health of the system and coordinates system-wide recovery procedures as neces-

sary. Monitors collect information from clients, SMART diagnostic tools, and NAD RPCs to gauge the status of the system. Monitors build on the per-object reconfiguration mechanisms described above to respond to events that individual clients don't address, such as load imbalance across the system, stores nearing capacity, and device failures.

Rebalance

Strata provides a rebalance facility which is capable of performing system-wide reconfiguration to repair broken replicas, prevent NADs from filling to capacity, and improve load distribution across NADs. This facility is in turn used to recover from device failures and expand onto new hardware.

Rebalance proceeds in two stages. In the first stage, the monitor retrieves the current system configuration, including the status of all NADs and virtual address map of every virtual object. It then constructs a new layout for the replicas according to a customizable placement policy. This process is scriptable and can be easily tailored to suit specific performance and durability requirements for individual deployments (see § 3.7.3 for some analysis of the effects of different placement policies). The default policy uses a greedy algorithm that considers a number of criteria designed to ensure that replicated physical objects do not share fault domains, capacity imbalances are avoided as much as possible, and migration overheads are kept reasonably low. The new layout is formulated as a rebalance plan describing what changes need to be applied to individual replica sets to achieve the desired configuration.

In the second stage, the monitor coordinates the execution of the rebalance plan by initiating resync tasks on individual NADs to effect the necessary data migration. When replicas need to be moved, the migration is performed in three steps:

1. A new replica is added to the destination NAD
2. A resync task is performed to transfer the data
3. The old replica is removed from the source NAD

This requires two reconfiguration events for the replica set, the first to extend it to include the new replica, and the second to prune the original after the resync has completed. The monitor coordinates this procedure across all NADs and clients for all modified virtual objects.

Device Failure

Strata determines that a NAD has failed either when it receives a hardware failure notification from a responsive NAD (such as a failed flash device or excessive error count) or when it observes that a NAD has stopped responding to requests for more than a configurable timeout. In either case, the monitor responds by taking the NAD offline and initiating a system-wide reconfiguration to repair redundancy.

The first thing the monitor does when taking a NAD offline is to disconnect it from the data path VLAN. This is a strong benefit of integrating directly against an Ethernet switch in our environment: prior to taking corrective action, the NAD is synchronously disconnected from the network for all request traffic, avoiding the distributed systems complexities that stem from things such as overloaded components appearing to fail and then returning long after a timeout in an inconsistent state. Rather than attempting to use completely end-host mechanisms such as watchdogs to trigger reboots, or agreement protocols to inform all clients of a NAD's failure, Strata disables the VLAN and requires that the failed NAD reconnect on the (separate) control VLAN in the event that it returns to life in the future.

From this point, the recovery logic is straight forward. The NAD is marked as failed in the configuration database and a rebalance job is initiated to repair any replica sets containing replicas on the failed NAD.

Elastic Scale Out

Strata responds to the introduction of new hardware much in the same way that it responds to failures. When the monitor observes that new hardware has been installed, it uses the rebalance facility to generate a layout that incorporates the

new devices. Because replication is generally configured underneath striping, we can migrate virtual objects at the granularity of individual stripes, allowing a single striped file to exploit the aggregated performance of many devices. Objects, whether whole files or individual stripes, can be moved to another NAD even while the file is online, using the existing resync mechanism. New NADs are populated in a controlled manner to limit the impact of background IO on active client workloads.

3.6 Storage Protocols

Strata supports legacy protocols by providing an execution runtime for hosting protocol servers. Protocols are built as thin presentation layers on top of the dispatch interfaces; multiple protocol instances can operate side by side. Implementations can also leverage SDN-based protocol scaling to transparently spread multiple clients across the distributed runtime environment.

3.6.1 Scalable NFS

Strata is designed so that application developers can focus primarily on implementing protocol specifications without worrying much about how to organize data on disk. We expect that many storage protocols can be implemented as thin wrappers around the provided dispatch library. Our NFS implementation, for example, maps very cleanly onto the high-level dispatch APIs, providing only protocol-specific extensions like RPC marshalling and NFS-style access control. It takes advantage of the configuration database to store mappings between the NFS namespace and the backend objects, and it relies exclusively on the striping and replication processors to implement the data path. Moreover, Strata allows NFS servers to be instantiated across multiple backend nodes, automatically distributing the additional processing overhead across backend compute resources.

3.6.2 SDN Protocol Scaling

Scaling legacy storage protocols can be challenging, especially when the protocols were not originally designed for a distributed back end. Protocol scalability limitations may not pose significant problems for traditional arrays, which already sit behind relatively narrow network interfaces, but they can become a performance bottleneck in Strata's distributed architecture.

A core property that limits scale of access bandwidth of conventional IP storage protocols is the presentation of storage servers behind a single IP address. Fortunately, emerging "software defined" network (SDN) switches provide interfaces that allow applications to take more precise control over packet forwarding through Ethernet switches than has traditionally been possible.

Using the OpenFlow protocol, a software controller is able to interact with the switch by pushing flow-specific rules onto the switch's forwarding path. OpenFlow rules are effectively wild-carded packet filters and associated actions that tell a switch what to do when a matching packet is identified. SDN switches (our implementation currently uses an Arista Networks 7050T-52) interpret these flow rules and push them down onto the switch's TCAM or L2/L3 forwarding tables.

By manipulating traffic through the switch at the granularity of individual flows, Strata protocol implementations are able to present a single logical IP address to multiple clients. Rules are installed on the switch to trigger a fault event whenever a new NFS session is opened, and the resulting exception path determines which protocol instance to forward that session to initially. A service monitors network activity and migrates client connections as necessary to maintain an even workload distribution.

The protocol scaling API wraps and extends the conventional socket API, allowing a protocol implementation to bind to and listen on a shared IP address across all of its instances. The client load balancer then monitors the traffic demands across all of these connections and initiates flow migration in response to overload on any individual physical connection.

In its simplest form, client migration is handled entirely at the transport layer. When the protocol load balancer observes that a specific NAD is overloaded, it updates the routing tables to redirect the busiest client workload to a different NAD. Once the client's traffic is diverted, it receives a TCP `RST` from the new NAD and establishes a new connection, thereby transparently migrating traffic to the new NAD.

Strata also provides hooks for situations where application layer coordination is required to make migration safe. For example, our NFS implementation registers a pre-migration routine with the load balancer, which allows the source NFS server to flush any pending, non-idempotent requests (such as `create` or `remove`) before the connection is redirected to the destination server.

3.7 Evaluation

In this section we evaluate our system both in terms of effective use of flash resources, and as a scalable, reliable provider of storage for NFS clients. First, we establish baseline performance over a traditional NFS server on the same hardware. Then we evaluate how performance scales as nodes are added and removed from the system, using VM-based workloads over the legacy NFS interface, which is oblivious to cluster changes. In addition, we compare the effects of load balancing and object placement policy on performance. We then test reliability in the face of node failure, which is a crucial feature of any distributed storage system. We also examine the relation between CPU power and performance in our system as a demonstration of the need to balance node power between flash, network and CPU.

3.7.1 Test Environment

Evaluation was performed on a cluster of the maximum size allowed by our 48-port switch: 12 NADs, each of which has two 10 gigabit Ethernet ports, two 800 GB Intel 910 PCIe flash cards, 6 3 TB SATA drives, 64 GB of RAM, and 2 Xen E5-2620 processors at 2 GHz with 6 cores/12 threads each, and 12 clients, in the form of

Server	Read IOPS	Write IOPS
Strata	40287	9960
KNFS	23377	5796

Table 3.1: Random IO performance on Strata versus KNFS

Dell PowerEdge R420 servers running ESXi 5.0, with two 10 gigabit ports each, 64 GB of RAM, and 2 Xeon E5-2470 processors at 2.3 GHz with 8 cores/16 threads each. We configured the deployment to maintain two replicas of every stored object, without striping (since it unnecessarily complicates placement comparisons and has little benefit for symmetric workloads). Garbage collection is active, and the deployment is in its standard configuration with a disk tier enabled, but the workloads have been configured to fit entirely within flash, as the effects of cache misses to magnetic media are not relevant to this chapter.

3.7.2 Baseline Performance

To provide some performance context for our architecture versus a typical NFS implementation, we compare two minimal deployments of NFS over flash. We set Strata to serve a single flash card, with no replication or striping, and mounted it loopback. We ran a FIO [14] workload with a 4K IO size 80/20 read-write mix at a queue depth of 128 against a fully allocated file. We then formatted the flash card with ext4, exported it with the linux kernel NFS server, and ran the same test. The results are in Table 3.1. As the table shows, we offer good NFS performance at the level of individual devices. In the following section we proceed to evaluate scalability.

3.7.3 Scalability

In this section we evaluate how well performance scales as we add NADs to the cluster. We begin each test by deploying 96 VMs (8 per client) into a cluster of 2 NADs. We choose this number of VMs because ESXi limits the queue depth for a VM to 32 outstanding requests, but we do not see maximum performance until a

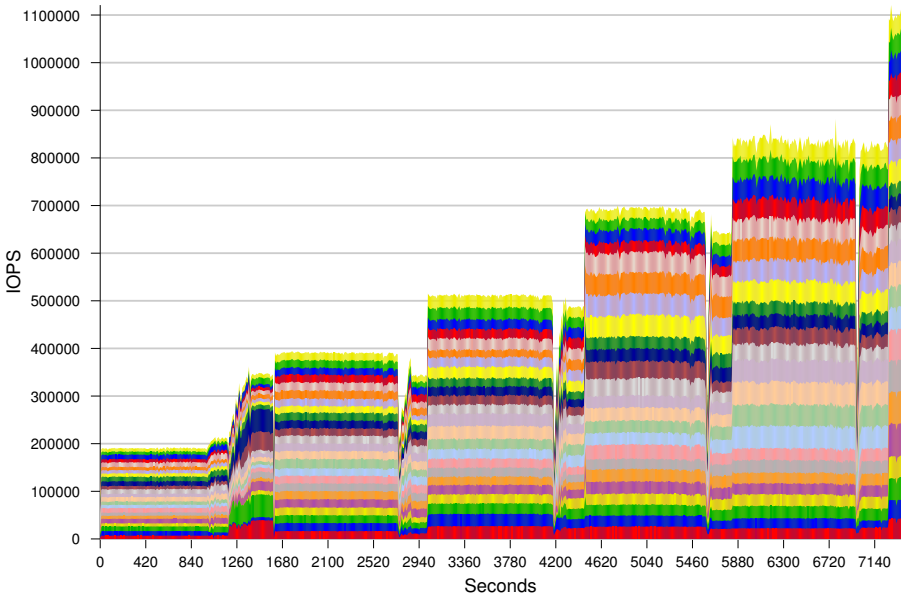


Figure 3.4: IOPS over time, read-only workload

queue depth of 128 per flash card. The VMs are each configured to run the same FIO workload for a given test. In Figure 3.4, FIO generates 4K random reads to focus on IOPS scalability. In Figure 3.5, FIO generates an 80/20 mix of reads and writes at 128K block size in a Pareto distribution such that 80% of requests go to 20% of the data. This is meant to be more representative of real VM workloads, but with enough offered load to completely saturate the cluster.

As the tests run, we periodically add NADs, two at a time, up to a maximum of twelve[2]. When each pair of NADs comes online, a rebalancing process automatically begins to move data across the cluster so that the amount of data on each NAD is balanced. When it completes, we run in a steady state for two minutes and then add the next pair. In both figures, the periods where rebalancing is in progress are reflected by a temporary drop in performance (as the rebalance process competes with client workloads for resources), followed by a rapid increase in overall performance when the new nodes are marked available, triggering the

[2]ten for the read/write test due to an unfortunate test harness problem

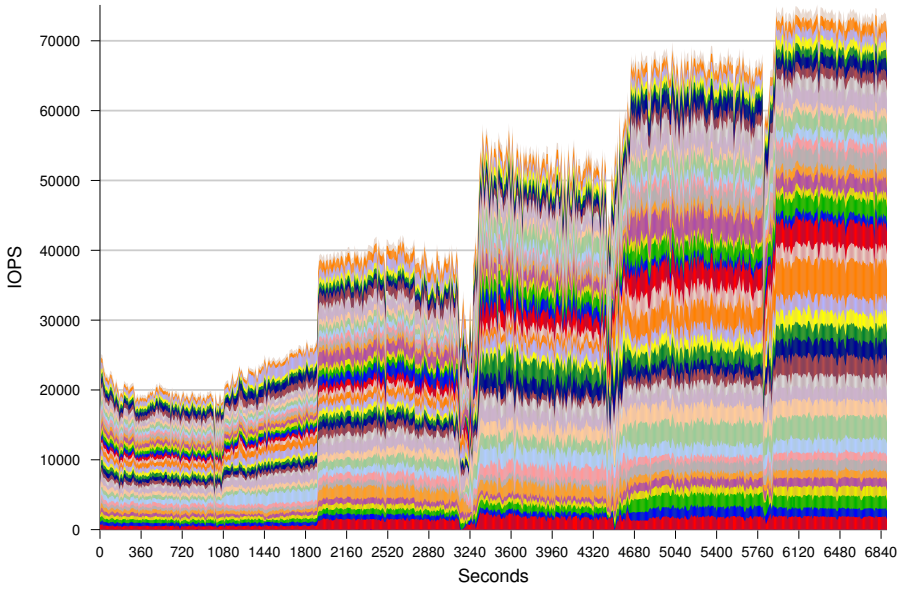


Figure 3.5: IOPS over time, 80/20 R/W workload

switch to load-balance clients to them. A cluster of 12 NADs achieves over 1 million IOPS in the IOPS test, and 10 NADs achieve 70,000 IOPS (representing more than 9 gigabytes/second of throughput) in the 80/20 test.

We also test the effect of placement and load balancing on overall performance. If the location of a workload source is unpredictable (as in a VM data center with virtual machine migration enabled), we need to be able to migrate clients quickly in response to load. However, if the configuration is more static or can be predicted in advance, we may benefit from attempting to place clients and data together to reduce the network overhead incurred by remote IO requests. As discussed in § 3.5.2, the load-balancing and data migration features of Strata make both approaches possible. Figure 3.4 is the result of an aggressive local placement policy, in which data is placed on the same NAD as its clients, and both are moved as the number of devices changes. This achieves the best possible performance at the cost of considerable data movement. In contrast, Figure 3.6 shows the performance of an otherwise identical test configuration when data is placed randomly (while

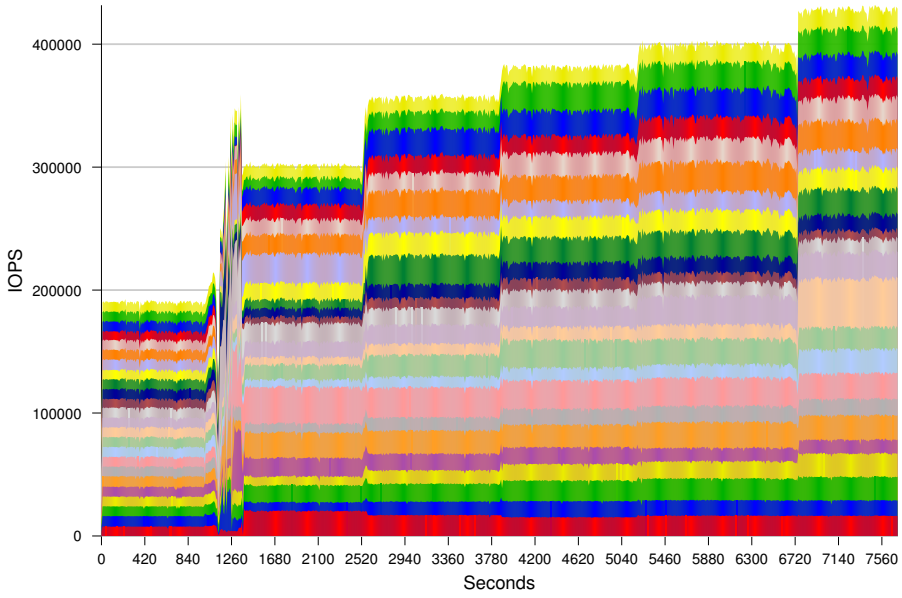


Figure 3.6: IOPS over time, read-only workload with random placement

still satisfying fault tolerance and even distribution constraints), rather than being moved according to client requests. The pareto workload (Figure 3.5) is also configured with the default random placement policy, which is the main reason that it does not scale linearly: as the number of nodes increases, so does the probability that a request will need to be forwarded to a remote NAD.

3.7.4 Node Failure

As a counterpoint to the scalability tests run in the previous section, we also tested the behaviour of the cluster when a node is lost. We configured a 10 NAD cluster with 10 clients hosting 4 VMs each, running the 80/20 Pareto workload described earlier. Figure 3.7 shows the behaviour of the system during this experiment. After the VMs had been running for a short time, we powered off one of the NADs by IPMI, waited 60 seconds, then powered it back on. During the node outage, the system continued to run uninterrupted but with lower throughput. When the node

came back up, it spent some time resynchronizing its objects to restore full replication to the system, and then rejoined the cluster. The client load balancer shifted clients onto it and throughput was restored (within the variance resulting from the client load balancer's placement decisions).

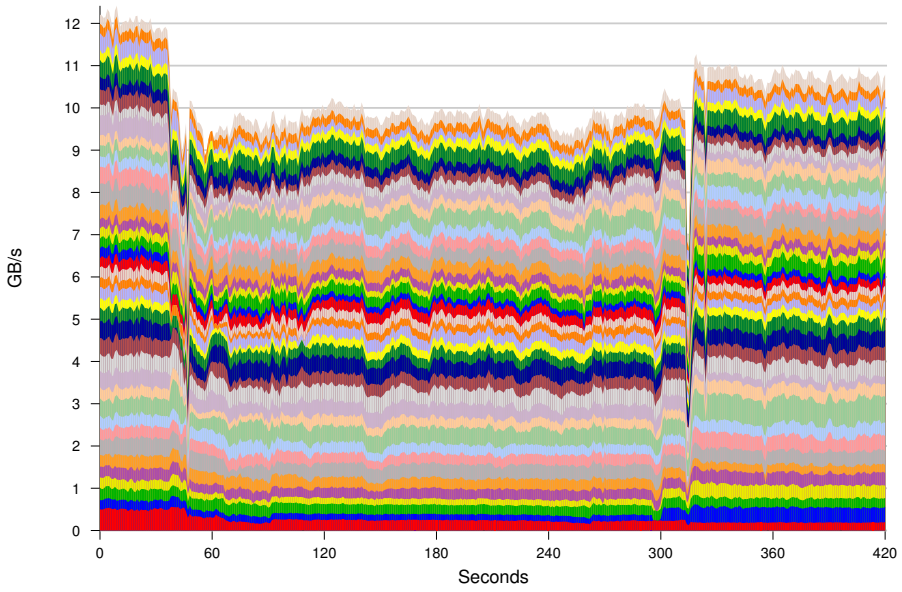


Figure 3.7: Aggregate bandwidth for 80/20 clients during failover and recovery

3.7.5 Protocol Overhead

The benchmarks up to this point have all been run inside VMs whose storage is provided by a virtual disk that Strata exports by NFS to ESXi. This configuration requires no changes on the part of the clients to scale across a cluster, but does impose overheads. To quantify these overheads we wrote a custom FIO engine that is capable of performing IO directly against our native dispatch interface (that is, the API by which our NFS protocol gateway interacts with the NADs). We then compared the performance of a single VM running a random 4k read FIO workload (for maximum possible IOPS) against a VMDK exported by NFS to the same workload run against our native dispatch engine. In this experiment, the VMDK-based exper-

CPU	IOPS	Freq (Cores)	Price
E5-2620	127K	2 GHz (6)	$406
E5-2640	153K (+*20%*)	2.5 GHz (6)	$885
E5-2650v2	188K (+*48%*)	2.6 GHz (8)	$1166
E5-2660v2	183K (+*44%*)	2.2 GHz (10)	$1389

Table 3.2: Achieved IOPS on an 80/20 random 4K workload across 2 NADs

iment produced an average of 50240 IOPS, whereas direct access achieved 54060 IOPS, for an improvement of roughly 8%.

3.7.6 Effect of CPU on Performance

A workload running at full throttle with small requests completely saturates the CPU. This remains true despite significant development effort in performance debugging, and a great many improvements to minimize data movement and contention. In this section we report the performance improvements resulting from faster CPUs. These results are from random 4K NFS requests in an 80/20 read-write mix at 128 queue depth over four 10Gb links to a cluster of two NADs, each equipped with 2 physical CPUs.

Table 3.2 shows the results of these tests. In short, it is possible to "buy" additional storage performance under full load by upgrading the CPUs into a more "balanced" configuration. The wins are significant and carry a non-trivial increase in the system cost. As a result of this experimentation, we elected to use a higher performance CPU in the shipping version of the product.

3.8 Related Work

Strata applies principles from prior work in server virtualization, both in the form of hypervisor [18, 118] and lib-OS [45] architectures, to solve the problem of sharing and scaling access to fast non-volatile memories among a heterogeneous set of clients. Our contributions build upon the efforts of existing research in several areas.

Recently, researchers have begin to investigate a broad range of system performance problems posed by storage class memory in single servers [15], including current PCIe flash devices [113], next generation PCM [5], and byte addressability [33]. Moneta [28] proposed solutions to an extensive set of performance bottlenecks over the PCIe bus interface to storage, and others have investigated improving the performance of storage class memory through polling [127], and avoiding system call overheads altogether [29]. We draw from this body of work to optimize the performance of our dispatch library, and use this baseline to deliver a high performance scale-out network storage service. In many cases, we would benefit further from these efforts—for example, our implementation could be optimized to offload per-object access control checks, as in Moneta-D [29]. There is also a body of work on efficiently using flash as a caching layer for slower, cheaper storage in the context of large file hosting. For example, S-CAVE [76] optimizes cache utilization on flash for multiple virtual machines on a single VMware host by running as a hypervisor module. This work is largely complementary to ours; we support using flash as a caching layer and would benefit from more effective cache management strategies.

Prior research into scale-out storage systems, such as FAWN [9], and Corfu [16] has considered the impact of a range of NV memory devices on cluster storage performance. However, to date these systems have been designed towards lightweight processors paired with simple flash devices. It is not clear that this balance is the correct one, as evidenced by the tendency to evaluate these same designs on significantly more powerful hardware platforms than they are intended to operate [16]. Strata is explicitly designed for dense virtualized server clusters backed by performance-dense PCIe-based nonvolatile memory. In addition, like older commodity disk-oriented systems including Petal [71, 112] and FAB [95], prior storage systems have tended to focus on building aggregation features at the lowest level of their designs, and then adding a single presentation layer on top. Strata in contrasts isolates shares each powerful PCIe-based storage class memory as its underlying primitive. This has allowed us to present a scalable runtime environment in which multiple protocols can coexist as peers without sacrificing the raw performance that today's high performance memory can provide. Many scale-out storage systems,

including NV-Heaps [32], Ceph/RADOS [115, 117], and even pNFS [56] are unable to support the legacy formats in enterprise environments. Our agnosticism to any particular protocol is similar to approach used by Ursa Minor [1], which also boasted a versatile client library protocol to share access to a cluster of magnetic disks.

Strata does not attempt to provide storage for datacenter-scale environments, unlike systems including Azure [25], FDS [88], or Bigtable [30]. Storage systems in this space differ significantly in their intended workload, as they emphasize high throughput linear operations. Strata's managed network would also need to be extended to support datacenter-sized scale out. We also differ from in-RAM approaches such a RAMCloud [91] and memcached [46], which offer a different class of durability guarantee and cost.

3.9 Conclusion

Storage system design faces a sea change resulting from the dramatic increase in the performance density of its component media. Distributed storage systems composed of even a small number of network-attached flash devices are now capable of matching the offered load of traditional systems that would have required multiple racks of spinning disks.

Strata is an enterprise storage architecture that responds to the performance characteristics of PCIe storage devices. Using building blocks of well-balanced flash, compute, and network resources and then pairing the design with the integration of SDN-based Ethernet switches, Strata provides an incrementally deployable, dynamically scalable storage system.

Strata's initial design is specifically targeted at enterprise deployments of VMware ESX, which is one of the dominant drivers of new storage deployments in enterprise environments today. The system achieves high performance and scalability for this specific NFS environment while allowing applications to interact directly with virtualized, network-attached flash hardware over new protocols. This is achieved by cleanly partitioning our storage implementation into an underly-

ing, low-overhead virtualization layer and a scalable framework for implementing storage protocols. Over the next year, we intend to extend the system to provide general-purpose NFS support by layering a scalable and distributed metadata service and small object support above the base layer of coarse-grained storage primitives.

Chapter 4

Characterizing Storage Workloads with Counter Stacks

A version of this chapter was published at the 11th USENIX Conference on Operating Systems Design and Implementation in 2014 [122].

4.1 Introduction

Caching is poorly understood. Despite being a pervasive element of computer system design – one that spans processor, storage system, operating system, and even application architecture – the effective sizing of memory tiers and the design of algorithms that place data within them remains an art of characterizing and approximating common case behaviors.

The design of hierarchical memories is complicated by two factors: First, the collection of live workload-specific data that might be analyzed to make "application aware" decisions is generally too expensive to be worthwhile. Approaches that model workloads to make placement decisions risk consuming the computational and memory resources that they are trying to preserve. As a result, systems in many domains have tended to use simple, general purpose algorithms such as LRU

to manage cache placement. Second, attempting to perform offline analysis of access patterns suffers from the performance overheads imposed in trace collection, and the practical challenges of both privacy and sheer volume, in sharing and analyzing access traces.

Today, these problems are especially pronounced in designing enterprise storage systems. Flash memories are now available in three considerably different form factors: as SAS or SATA-attached solid state disks, as NVMe devices connected over the PCIe bus, and finally as flash-backed nonvolatile RAM, accessible over a DIMM interface. These three connectivity models all use the same underlying flash memory, but present performance and pricing that are pairwise 1-2 orders of magnitude apart. Further, in addition to solid-state memories, spinning disks remain an economical option for the storage of cold data.

This chapter describes an approach to modeling, analyzing, and reasoning about memory access patterns that has been motivated through our experience in designing a hierarchical storage system [34] that combines these varying classes of storage media. The system is a scalable, network-attached storage system that can benefit from workload awareness in two ways: First, the system can manage allocation of the memory hierarchy in response to workload characteristics. Second, the capacity at each level of the hierarchy can be independently expanded to satisfy application demands, by adding additional hardware. Both of these properties require a more precise ability to understand and characterize individual storage workloads, and in particular their working set sizes over time.

Miss ratio curves (MRCs) are an effective tool for assessing working set sizes, but the space and time required to generate them make them impractical for large-scale storage workloads. We present a new data structure, the *counter stack*, which can generate approximate LRU MRCs in sublinear space, for the first time making this type of analysis feasible in the storage domain.

Counter stacks use probabilistic counters [47] to estimate LRU MRCs. The original approach to generating MRCs is based on the observation that a block's 'stack distance' (also known as its 'reuse distance') gives the capacity needed to cache it, and this distance is exactly the number of unique blocks accessed since the previ-

ous request for the block. The key idea behind counter stacks is that probabilistic counters can be used to efficiently estimate stack distances, allowing us to compute approximate MRCs at a fraction of the cost of traditional techniques.

Counter stacks are fast. Our Java implementation can process a week-long trace of 13 enterprise servers in 17 minutes using just 80 MB of RAM; at a rate of 2.3 million requests per second, the approach is practical for online analysis in production systems. By comparison, a recent C implementation of a tree-based optimization [89] of Mattson's original stack algorithm [78] takes roughly an hour and 92 GB of RAM to process the same trace.

Our contributions in this chapter are threefold. First, we introduce a novel technique for estimating miss ratio curves using counter stacks, and we evaluate the performance and accuracy of this technique. Second, we show how counter stacks can be periodically checkpointed and streamed to disk to provide a highly compressed representation of storage workloads. Counter stack *streams* capture important details that are discarded by statistical aggregation while at the same time requiring orders of magnitude less storage and processing overhead than full request traces; a counter stack stream of the compressed 2.9 GB trace mentioned above consumes just 11 MB. Third, we present techniques for working with multiple independent counter stacks to estimate miss ratio curves for new workload combinations. Our library implements *slice*, *shift*, and *join* operations, enabling the nearly-instantaneous computation of MRCs for arbitrary workload combinations over arbitrary windows in time. These capabilities extend the functionality of MRC analysis and provide valuable insight into live workloads, as we demonstrate with a number of case studies.

4.2 Background

The many reporting facilities embedded in the modern Linux storage stack [21, 23, 61, 83] are testament to the importance of being able to accurately characterize live workloads. Common characterizations typically fall into one of two categories: coarse-grain aggregate statistics and full request traces. While these representa-

tions have their uses, they can be problematic for a number of reasons: averages and histograms discard key temporal information; sampling is vulnerable to the often bursty and irregular nature of storage workloads; and full traces impose impractical storage and processing overheads. New representations are needed which preserve the important features of full traces while remaining manageable to collect, store, and query.

Working set theory [36] provides a useful abstraction for describing workloads more concisely, particularly with respect to how they will behave in hierarchical memory systems. In the original formulation, working sets were defined as the set of all pages accessed by a process over a given epoch. This was later refined by using LRU modelling to derive an MRC for a given workload and restricting the working set to only those pages that exhibit strong locality. Characterizing workloads in terms of the unique, 'hot' pages they access makes it easier to understand their individual hardware requirements, and has proven useful in CPU cache management for many years [68, 93, 109]. These concepts hold for storage workloads as well, but their application in this domain is challenging for two reasons.

First, until now it has been prohibitively expensive to calculate the working set of storage workloads due to their large sizes. Mattson's original stack algorithm [78] required $O(NM)$ time and $O(M)$ space for a trace of N requests and M unique elements. An optimization using a balanced tree to maintain stack distances [7] reduces the time complexity to $O(N \log M)$, and recent approximation techniques [38, 126] reduce the time complexity even further, but they still have $O(M)$ space overheads, making them impractical for storage workloads that may contain billions of unique blocks.

Second, the extended duration of storage workloads leads to subtleties when reasoning about their working sets. CPU workloads are relatively short-lived, and in many cases it is sufficient to consider their working sets over small time intervals (e.g., a scheduling quantum) [132]. Storage workloads, on the other hand, can span weeks or months and can change dramatically over time. MRCs at this scale can be tricky: if they include too little history they may fail to capture important recurring patterns, but if they include too much history they can significantly misrepresent

recent behavior.

This phenomenon is further exacerbated by the fact that storage workloads already sit behind a file system cache and thus typically exhibit longer reuse distances than CPU workloads [133]. Consequently, cache misses in storage workloads may have a more pronounced effect on miss ratios than CPU cache misses, because subsequent re-accesses are likely to be absorbed by the file system cache rather than contributing to hits at the storage layer.

One implication of this is that MRC analysis needs to be performed over various time intervals to be effective in the storage domain. A workload's MRC over the past hour may differ dramatically from its MRC over the past day; both data points are useful, but neither provides a complete picture on its own.

This leads naturally to the notion of a *history of locality*: a workload representation which characterizes working sets as they change over time. Ideally, this representation contains enough information to produce MRCs over arbitrary ranges in time, in much the same way that full traces support statistical aggregation over arbitrary intervals. A naïve implementation could produce this representation by periodically instantiating new Mattson stacks at fixed intervals of a trace, thereby modelling independent LRU caches with various amounts of history, but such an approach would be impractical for real-world workloads.

In the following section we describe a novel technique for computing stack distances (and by extension, MRCs), from an inefficient, idealized form of counter stacks. § 4.4 explains several optimizations which allow a practical counter stack implementation that requires sublinear space, and § 4.5 presents the additional operations that counter stacks support, such as slicing and joining.

4.3 Counter Stacks

Counter stacks capture locality properties of a sequence of accesses within an address space. In the context of a storage system, accesses are typically read or write requests to physical disks, logical volumes, or individual files. A counter stack can

process a sequence of requests as they occur in a live storage system, or it can process, in a single pass, a trace of a storage workload. The purpose of a counter stack is to represent specific characteristics of the stream of requests in a form that is efficient to compute and store, and that preserves enough information to characterize aspects of the workload, such as cache behaviour.

Rather than representing a trace as a sequence of requests for specific addresses, counter stacks maintain a list of counters, which are periodically instantiated while processing the trace. Each counter records the number of *unique* trace elements observed since the inception of that counter; this captures the size of the working set over the corresponding portion of the trace. Computing and storing samples of working set size, rather than a complete access trace, yields a very compact representation of the trace that nevertheless reveals several useful properties, such as the number of unique blocks requested, or the stack distances of all requests, or phase changes in the working set. These properties enable computation of MRCs over arbitrary portions of the trace. Furthermore, this approach supports composition and extraction operations, such as joining together multiple traces or slicing traces by time, while examining only the compact representation, not the original traces.

4.3.1 Definition

A counter stack is an in-memory data structure that is updated while processing a trace. At each time step, the counter stack can report a list of values giving the numbers of distinct blocks that were requested between the current time and *all previous* points in time. This data structure evolves over time, and it is convenient to display its history as a matrix, in which each column records the values reported by the counter stack at some point in time.

Formally, given a trace sequence $(e_1 \dots e_N)$, where e_i is the ith trace element, consider an $N \times N$ matrix C whose entry in the ith row and jth column is the number of distinct elements in the set $\{e_i \dots e_j\}$. For example, the trace (a,b,c,a) yields the following matrix.

(	a,	b,	c,	a,	)
	1	2	3	3	
		1	2	3	
			1	2	
				1	

The j^{th} column of this matrix gives the values reported by the counter stack at time step j, i.e., the numbers of distinct blocks that were requested between that time and all previous times. The i^{th} row of the matrix can be viewed as the sequence of values produced by the counter that was instantiated at time step i.

The in-memory counter stack only stores enough information to produce, at any point in time, a single column of the matrix. To compute our desired properties over arbitrary portions of the trace, we need to store the entire history of the data structure, i.e., the entire matrix. However, the history does not need be stored in memory. Instead, at each time step we write to disk the current column of values reported by the counter stack. This can be viewed as checkpointing, or incrementally updating, the on-disk representation of the matrix. This on-disk representation is called a *counter stack stream*; for conciseness we will typically refer to it simply as a *stream*.

4.3.2 LRU Stack Distances

Stack distances and MRCs have numerous applications in cache sizing [78], memory partitioning between processes or VMs [62, 107, 109, 132], garbage collection frequency [128], program analysis [38, 131], workload phase detection [102], etc. A significant obstacle to the widespread use of MRCs is the cost of computing them, particularly the high storage cost [20, 89, 103, 106, 129] – all existing methods require linear space. Counter stacks eliminate this obstacle by providinge extremely efficient MRC computation while using sublinear space.

In this subsection we explain how stack distances, and hence MRCs, can be derived from counter stack streams. Recall that the stack distance of a given request is the number of distinct elements observed since the last reference to the requested element. Because a counter stack stores information about distinct elements, de-

termining the stack distance is straightforward. At time step j one must find the last position in the trace, i, of the requested element, then examine entry C_{ij} of the matrix to determine the number of distinct elements requested between times i and j. For example, let us consider the matrix given in § 4.3.1. To determine the stack distance for the second reference to trace element a at position 4, whose previous reference was at position 1, we look up the value $C_{1,4}$ and get a stack distance of 3.

This straightforward method ignores a subtlety: how can one find the last position in the trace of the requested element? It turns out that this information is implicitly contained in the counter stack. To explain this, suppose that the counter that was instantiated at time i does not increase during the processing of element e_j. Since this counter reports the number of *distinct* elements that it has seen, we can infer that this counter has already seen element e_j. On the other hand, if the counter instantiated at time $i+1$ does increase while processing e_j, then we can infer that this counter has not yet seen element e_j. Combining those inferences, we can conclude that i is the position of last reference.

These observations lead to a finite-differencing scheme that can pinpoint the positions of last reference. At each time step, we must determine how much each counter increases during the processing of the current element of the trace. This is called the *intra-counter* change, and it is defined to be

$$\Delta x_{ij} \;=\; C_{i,j} - C_{i,j-1}$$

To pinpoint the position of last reference, we must find the newest counter that does not increase. This can be done by comparing the intra-counter change of adjacent counters. This difference is called the *inter-counter* change, and it is defined to be

$$\Delta y_{ij} \;=\; \begin{cases} \Delta x_{i+1,j} - \Delta x_{i,j} & \text{if } i < j \\ 0 & \text{if } i = j \end{cases}$$

Let us illustrate these definitions with an example. Restricting our focus to the first four elements of the example trace from § 4.3.1, the matrices Δx and Δy are

{	a,	b,	c,	**a**	}
	1	1	1	0	
		1	1	1	
			1	1	
				1	

Δx

{	a,	b,	c,	**a**	}
	0	0	0	**1**	
		0	0	0	
			0	0	
				0	

Δy

Every column of Δy either contains only zeros, or contains a single 1. The former case occurs when the element requested in this column has never been requested before. In the latter case, if the single 1 appears in row i, then the last request for that element was at time i. For example, because $\Delta y_{14} = 1$, the last request for element a before time 4 was at time 1.

Determining the stack distance is now simple, as before. While processing column j of the stream, we infer that the last request for the element e_j occurred at time i by observing that $\Delta y_{ij} = 1$. The stack distance for the j^{th} request is the number of distinct elements that were requested between time i and time j, which is C_{ij}. Recall that the MRC at cache size x is the fraction of requests with stack distance exceeding x. Therefore given all the stack distances, we can easily compute the MRC.

4.4 Practical Counter Stacks

The idealized counter stack stream defined in § 4.3 stores the entire matrix C, so it requires space that is quadratic in the length of the trace. This is actually *more* expensive than storing the original trace. In this section we introduce several ideas that allow us to dramatically reduce the space of counter stacks and streams.

§ 4.4.1 discusses the natural idea of decreasing the time resolution, i.e., keeping only every d^{th} row and column of the matrix C. § 4.4.2 discusses the idea of pruning: eventually a counter may have observed the same set of elements as its adjacent counter, at which point maintaining both of them becomes unnecessary. Finally, § 4.4.3 introduces the crucial idea of using probabilistic counters to efficiently and compactly estimate the number of distinct elements seen in the trace.

4.4.1 Downsampling

The simplest way to improve the space used by counter stacks and streams is to decrease the time resolution. This idea is not novel, and similar techniques have been used in previous work [42].

In our context, decreasing the time resolution amounts to keeping only a small submatrix of C that provides enough data, and of sufficient accuracy, to be useful for applications. For example, one could start a new counter only at every d^{th} position in the trace; this amounts to keeping only every d^{th} row of the matrix C. Next, one could update the counters only at every d^{th} position in the trace; this amounts to keeping only every d^{th} column of the matrix C. We call this process *downsampling*.

Adjacent entries in the original matrix C can differ only by 1, so adjacent entries in the downsampled matrix can differ only by d. Thus, any entry that is missing from the downsampled matrix can be estimated using nearby entries that are present, up to additive error d. For large-scale workloads with billions of distinct elements, even choosing a very large value of d has negligible impact on the estimated stack distances and MRCs.

Our implementation uses a slightly more elaborate form of downsampling because we wish to combine traces that may have activity bursts in disjoint time intervals and avoid writing columns during idle periods. As well as starting a new counter and updating the old counters after every d^{th} request, we also start a new counter and update the old counters every s seconds with one exception: we do not output a column if the previous s seconds contain no activity. Our experiments reported in § 4.7 pick $d = 10^6$ and $s \in \{60, 3600\}$.

4.4.2 Pruning

Recall that every row of the matrix contains a sequence of values reported by some counter. For any two adjacent counters, the older one (the upper row) will always emit values larger than or equal to the younger one (the lower row). Let us consider

the difference of these counters. Initially, at the time the younger one is created, their difference is simply the number of distinct elements seen by the older counter so far. If any of these elements reappears in the trace, the older counter will not increase (as it has seen this element before), but the younger counter will increase, so the difference of the counters shrinks.

If at some point the younger counter has seen every element seen by the older counter, then their difference becomes zero and will remain zero forever. In this case, the younger counter provides no additional information, so it can be deleted. An extension of this idea is that, when the difference between the counters becomes sufficiently small, the younger counter provides negligible additional information. In this case, the younger counter can again be deleted, and its value can be approximated by referring to the older counter. We call this process *pruning*.

The simplest pruning strategy is to delete the younger counter whenever its value differs from its older neighbor by at most p. This strategy ensures that the number of active counters at any point in time is at most M/p. (Recall that M is the number of distinct blocks in the entire trace.) In our current implementation, in order to fix a set of parameters that work well across many workloads of varying sizes, we instead delete the younger counter whenever its value is at least $(1-\delta)$ times the older counter's value. This ensures that the number of active counters is at most $O(\log(M)/\delta)$. Our experiments reported in § 4.7 pick $\delta \in \{0.1, 0.02\}$.

4.4.3 Probabilistic Counters

Counter stack streams contain the number of distinct blocks seen in the trace between any two points in time (neglecting the effects of downsampling and pruning). The on-disk stream only needs to store this matrix of counts, as the examples in § 4.3 suggested. The in-memory counter stack has a more difficult job – it must be able to update these counts while processing the trace, so each counter must keep an internal representation of the set of blocks it has seen.

The naïve approach is for each counter to represent this set explicitly, but this would require quadratic memory usage (again, neglecting downsampling and pruning).

A slight improvement can be obtained through the use of Bloom filters [22], but for an acceptable error tolerance, the space would still be prohibitively large. Our approach is to use a tool, called a *probabilistic counter* or *cardinality estimator*, that was developed over the past thirty years in the streaming algorithms and database communities.

Probabilistic counters consume extremely little space and have guaranteed accuracy. The most practical of these is the HyperLogLog counter [47], which we use in our implementation. Each count appearing in our on-disk stream is not the true count of distinct blocks, but rather an estimate produced by a HyperLogLog counter which is correct up to multiplicative factor $1+\varepsilon$. The memory usage of each HyperLogLog counter is roughly *logarithmic* in M, with more accurate counters requiring more space. More concretely, our evaluation discussed in § 4.7 uses as little as 53 MB of memory to process traces containing over a hundred million requests and distinct blocks.

4.4.4 LRU Stack Distances

The technique in § 4.3.2 for computing stack distances and MRCs using idealized counter stacks can be adapted to use practical counter stacks. The matrices Δx and Δy are defined as before, but are now based on the downsampled, pruned matrix containing probabilistic counts. Previously we asserted that every column of Δy is either all zeros or contains a single 1. This is no longer true. The entry Δy_{ij} now reports the *number* of requests since the counters were last updated whose stack distance was approximately C_{ij}.

To approximate the stack distances of all requests, we process all columns of the stream. As there may be many non-zero entries in the j^{th} column of Δy, we record Δy_{ij} occurrences of stack distance C_{ij} for every i. As before, given all stack distances we can compute the MRC.

An online version of this approach which does not emit streams can produce an MRC of guaranteed accuracy using provably sublinear memory. In a companion paper [41] we prove the following theorem. The key point is that the space depends

polynomially on ℓ and ε, the parameters controlling the precision of the MRC, but only logarithmically on N, the length of the trace.

Theorem 1 *The online algorithm produces an estimated* MRC *that is correct to within additive error* ε *at cache sizes* $\frac{1}{\ell}M, \frac{2}{\ell}M, \frac{3}{\ell}M, \ldots, M$ *using only* $O(\ell^2 \log(M) \log^2(N)/\varepsilon^2)$ *bits of space, with high probability.*

4.5 The Counter Stack API

The previous two sections have given an abstract view of counter stacks. In this section we describe the system that we have implemented based on those ideas. The system is a flexible, memory-efficient library that can be used to process traces, produce counter stack streams, and perform queries on those streams. The workflow of applications that use this library is illustrated in Figure 4.1.

4.5.1 On-disk Streams

The on-disk streams output by the library are produced by periodically outputting a new column of the matrix. As discussed in § 4.4, a new column is produced if either d requests have been observed in the trace or s seconds have elapsed (in the trace's time) since the last column was produced, except for idle periods, which are elided. Each column is written to disk in a sparse format to incorporate the fact that pruning may cause numerous entries to be missing.

In addition, the on-disk matrix C includes an extra row, called row R, which records the raw number of requests observed in the stream. That is, C_{Rj} contains the total number of requests processed at the time that the j^{th} column is output. Finally, the on-disk stream also records the trace's time of the current request.

4.5.2 Compute Queries

The counter stack library supports three computational queries on streams: *Request Count*, *Unique Request Count* and *MRC*.

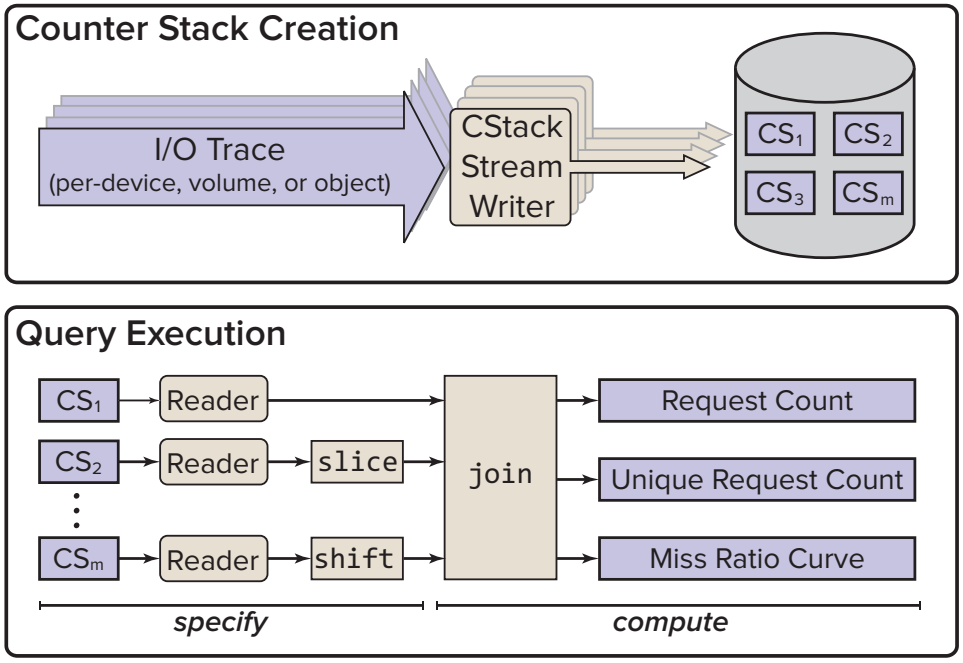


Figure 4.1: The counter stack library architecture

The first two query operations are straightforward but useful, as we will show in § 4.8.4. The Request Count query simply asks for the total number of requests that occur in the stream, which is C_{Rj} where j is the index of the last column. The Unique Request Count query is similar except that it asks for the total number of unique requests, which is C_{1j}.

The most complicated stream operation is the MRC query, which asks for the miss ratio curve of the given stream. This query is processed using the method described in § 4.4.4.

4.5.3 Time Slicing and Shifting

It is often useful to analyze only a subset of a given trace within a specific time interval. We refer to this time-based selection as *slicing*. It is similarly useful when joining traces to alter the time signature by a constant time interval. We refer to

this alteration as *shifting*.

The counter stack library supports slicing and shifting as specification operations. Given a stream containing a matrix C, the stream for the time slice between time step i and j is the submatrix with corners at C_{ii} and C_{jj}. Likewise, to obtain the stream for the trace shifted forward/backward s time units, we simply add/subtract s to each of the time indices associated with the rows and columns of the matrix.

4.5.4 Joining

Given two or more workloads, it is often useful to understand the behavior that would result if they were combined into a single workload. For example, if each workload is an IO trace of a different process, one may want to investigate the cache performance of those processes with a shared LRU cache.

Counter stacks enable such analyses through the *join* operation. Given two counter stack streams, the desired output of the join operation is what one would obtain by merging the original two traces according to the traces' times, then producing a new counter stack stream from that merged trace. Our library can produce this new stream using only the two given streams, without examining the original traces. The only assumption we require is that the two streams must access disjoint sets of blocks.

The join process would be simple if, for every i, the time of the i^{th} request were the same in both traces; in this case, we could simply add the matrices stored in the two streams. Unfortunately that assumption is implausible, so more effort is required. The main ideas are to:

- *Expand* the two matrices so that each has a row and column for every time that appears in either trace.
- *Interpolate* to fill in the new matrix entries.
- *Add* the resulting matrices together.

Let us illustrate this process with an example. Consider a trace **A** that requests

time	1:00	1:02	1:05	1:14	1:17
A	a		b		b
$C_{\mathbf{A}}$	1	**1**	2	**2**	2
		0	**1**	**1**	**1**
			1	**1**	1
				0	**1**
					1
B		d		d	
$C_{\mathbf{B}}$	**0**	**1**	**1**	**1**	**1**
		1	**1**	1	**1**
			0	**1**	**1**
				1	**1**
					0
merge	a	d	b	d	b
$C_{\mathbf{A}} + C_{\mathbf{B}}$	1	2	3	3	3
		1	2	2	2
			1	2	2
				1	2
					1

Figure 4.2: An example illustrating the join operation

blocks (a,b,b) at times 1:00, 1:05, 1:17, and a trace **B** requests blocks (d,d) at times 1:02 and 1:14. The merge of the two traces is as follows:

time	1:00	1:02	1:05	1:14	1:17
A	a		b		b
B		d		d	
merge	a	d	b	d	b

To join these streams, we must expand the matrices in the two streams so that each has five rows and columns, corresponding to the five times that appear in the traces. After this expansion, each matrix is missing entries corresponding to times that were missing in its trace. We fill in those missing entries by an interpolation process: a missing row is filled by copying the nearest row beneath it, and a missing column is filled by copying the nearest column to the left of it. Figure 4.2 shows the resulting matrices; interpolated values are shown in bold blue.

Pruned counters can sometimes create negative values in Δx. For example, after pruning a counter in row j at time t, the interpolated value of the pruned counter at

$t+1$ is set to the nearest row beneath it, representing a younger counter. Often, this lower counter has a smaller value than the pruned counter. The interpolated value at $t+1$ will then be less than its previous value at t, producing a negative intra-counter change. We can avoid introducing negative values in Δx by replacing any negative values in Δx by the nearest nonnegative value beneath it. This replacement has the same effect of changing the value of the pruned counter to the lower counter in column t prior to calculating the intra-counter change for the column representing $t+1$.

4.6 Error and Uncertainty

While each of the optimizations described in § 4.4 dramatically reduce the storage requirements of counter stacks, they may also introduce uncertainty and error into the final calculations. In this section, we discuss potential sources of error, as well as how to modify the different operations described in § 4.3 to compute lower and upper bounds on the stack distances.

4.6.1 Counter Error

HyperLogLog counters introduce error in two ways: count estimation and simultaneous register updates. HyperLogLog counters report a count of distinct elements that is only correct up to multiplicative factor ε, which is determined by a precision parameter. This uncertainty produces deviation from the true MRC and can be controlled by increasing the precision of the HyperLogLog counters, at the cost of a greater memory requirement.

Simultaneous register updates introduce a subtler form of error. A HyperLogLog counter estimates unique counts by taking the harmonic mean of a set of internal variables called *registers*. Due to the design of HLLs, sometimes a register update might cause the older counter to increase in value more than the younger counter. This phonemoneon leads to negative updates in Δy, because older counters are expected to change more slowly than younger counters. Theorem 1 implies that

the negative entries in the Δy matrix introduced by simultaneous register updates are offset by corresponding over-estimates when register modifications between counters are not simultaneous.

In some cases, the histogram of stack distances may accumulate enough negative entries that there are bins with negative counts. The cumulative sum of such a histogram will result in a non-monotonic MRC. We can enforce a monotonic MRC by accumulating any negative histogram bins in a separate counter, carrying the difference forward in the cumulative sum and discounting positive bins by the negative count. In practice, negative histogram entries make up less then one percent of the reported stack distances, with little to no visible effect on the accumulated MRC.

4.6.2 Downsampling Uncertainty

Whereas the scheme of § 4.3.2 computes stack distances exactly, the modified scheme of § 4.4.4 only computes approximations. This uncertainty in the stack distances is caused by downsampling, pruning and use of probabilistic counters. To illustrate this, consider the example shown in Figure 4.3, and for simplicity let us ignore pruning and any probabilistic error.

At every time step j, the finite differencing scheme uses the matrix Δy to help estimate the stack distances for all requests that occurred since time step $j-1$. More concretely, if such a request increases the $(i+1)^{\text{th}}$ counter but does not increase the i^{th} counter, then we know that the most recent occurrence of the requested block lies somewhere between time step i and time step $i+1$. Since there may have been many requests between time i and time $i+1$, we do not have enough information to determine the stack distance exactly, but we estimate it up to additive error d (the downsampling factor). A careful analysis can show that the request must have stack distance at least $C_{i+1,j-1}+1$ and at most C_{ij}.

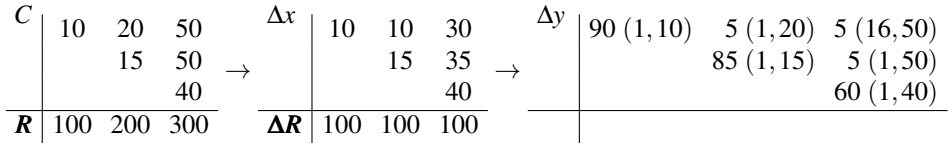


Figure 4.3: An example of computing stack distances using a downsampled matrix. The entries of Δy show the number of requests and the parenthesized values show the bounds on the stack distances that we can infer for those requests.

4.7 Evaluation

In this section we empirically validate two claims: (1) the time and space requirements of counter stack processing are sufficiently low that it can be used for online analysis of real storage workloads, and (2) the technique produces accurate, meaningful results.

We use a well-studied collection of storage traces released by Microsoft Research in Cambridge (MSR) [86] for much of our evaluation. The MSR traces record the disk activity (captured beneath the file system cache) of 13 servers with a combined total of 36 volumes. Notable workloads include a web proxy (`prxy`), a filer serving project directories (`proj`), a pair of source control servers (`src1` and `src2`), and a web server (`web`). The raw traces comprise 417 million records and consume just over 5 GB in compressed CSV format.

We compare our technique to the 'ground truth' obtained from full trace analysis (using *trace trees*, the tree-based optimization of Mattson's algorithm [78, 89]), and, where applicable, to a recent approximation technique [125] which derives estimated MRCs from *average footprints* (see § 4.9 for more details). For fairness, we modify the original implementation [37] by using a sparse dictionary to reduce memory overhead.

4.7.1 Performance

The following experiments were conducted on a Dell PowerEdge R720 with two six-core Intel Xeon processors and 96 GB of RAM. Traces were read from high-

Fidelity	Time	Memory	Throughput	Storage
low	17.10 m	78.5 MB	2.31M reqs/sec	747 KB
high	17.24 m	80.6 MB	2.29M reqs/sec	11 MB

Table 4.1: The resources required to create low and high fidelity counter stacks for the combined MSR workload (64 MB heap)

performance flash to eliminate disk IO bottlenecks.

Throughout this section we present figures for both 'low' and 'high' fidelity streams. We control the fidelity by adjusting the number of counters maintained in each stream; the parameters used in these experiments represent just two points of a wide spectrum, and were chosen in part to illustrate how accuracy can be traded for performance to meet individual needs.

We first report the resources required to convert a raw storage trace to a counter stack stream. The memory footprint for the conversion process is quite modest: converting the entire set of MSR traces to high-fidelity counter stacks can be done with about 80 MB of RAM [1]. The processing time is low as well: our Java implementation can convert a trace to a high-fidelity stream at a rate of 2.3 million requests per second with a 64 MB heap and 2.7 million requests per second with a 256 MB heap.

The size of counter stack streams can also be controlled by adjusting fidelity. Ignoring write requests, the full MSR workload consumes 2.9 GB in a compressed, binary format. We can reduce this to 854 MB by discarding latency values and capping timestamp resolutions at one second, and we can shave off another 50 MB through domain-specific compaction techniques like delta-encoding time and offset values. But as Table 4.1 shows, this is more than 70 times larger than a high-fidelity counter stack representation.

The compression achieved by counter stack streams is workload-dependent. High-

[1] This is not a lower bound. Additional reductions can be achieved at the expense of increased garbage collection activity in the JVM; for example, enforcing a heap limit of 32 MB increases processing time for the high-fidelity counter stack by about 30% and results in a peak resident set size of 53 MB.

fidelity streams of the MSR workloads are anywhere from 12 (`hm`) to 1,024 (`prxy`) times smaller than their compressed binary counterparts, with larger traces tending to compress better. A stream of the combined traces consumes just over 1.5 MB per day, meaning that weeks or even months of workload history can be retained at very reasonable storage costs.

Once a trace has been converted to a counter stack stream, performing queries is very quick. For example, an MRC for the entire week-long MSR trace can be computed from the counter stack stream in just seconds, with negligible memory overheads. By comparison, computing the same MRC using a trace tree takes about an hour and reaches a peak memory consumption of 92 GB, while the average footprint technique requires 8 and a half minutes and 23 GB of RAM.

4.7.2 Accuracy

Figure 4.4 shows miss ratio curves for each of the individual workloads contained in the MSR traces as well as the combined `master` trace; superimposed on the baseline curves (showing the exact MRCs) are the curves computed using footprint averages and counter stacks. Some of the workloads feature MRCs that are notably different from the convex functions assumed in the past [109]. The `web` workload is the most obvious example of this, and it is also the workload which causes the most trouble for the average footprint technique.

Figure 4.5 shows three examples of MRCs produced by *joining* individual counter stacks. The choice of workloads is somewhat arbitrary; we elected to join workloads of commensurate size so that each would contribute equally to the resulting merged MRC. As described in § 4.5.4, the join operation can introduce additional uncertainty due to the need to infer the values of missing counters, but the effects are not prominent with the high-fidelity counter stacks used in these examples.

We performed an analysis of curve errors at different fidelities, with `verylow` ($\delta = 0.46$, $d = 19M$, $s = 32K$) at one extreme and `high` ($\delta = 0.01$, $d = 1M$, $s = 60$) at the other. To measure curve error, we use the Mean Absolute Error (MAE) between a given curve and its ground-truth counterpart. The MAE is defined as the average

absolute difference between two series *mrc* and *mrc′*, or $\frac{1}{N}\sum|mrc(x) - mrc'(x)|$. Because MRCs range between 0 and 1, the MAEs are also confined to the same range, where a value of 0 implies perfectly corresponding curves. At the other extreme, it is difficult to know what constitutes a "bad" MAE because it is unlikely to be close to 1 except in singular cases. For example, the MAE between the `hm` and the `ts` Mattson curves is only 0.15. For the high fidelity counter stacks, we observe MAEs between 0.002 and 0.02, and for the average footprint algorithm, we observe MAEs between 0.001 and 0.04.

We find that curve error under compression is highly workload-dependent. We observed the largest errors on "jagged" workloads with sharp discontinuities, such as `src1` and `web`, while workloads with "flatter" MRCs such as `stg` and `usr` are almost invariant to compression. Figure 4.6 summarizes our findings on two such workloads. On the left, we illustrate the difference in the change in error as fidelity decreases for a jagged workload, `src1`, and a flat workload, `usr`. On the right, we show the smoothing effect of decreasing the counter stack fidelity by comparing the `verylow` and `high` fidelity curves against Mattson on `src1`.

4.8 Workload Analysis

We have shown that counter stacks can be used to produce accurate MRC estimations in a fraction of the time and space used by existing techniques. We now demonstrate some of the capabilities of the counter stack query interface through a series of case studies of the MSR traces.

4.8.1 Combined Workloads

Hit rates are often used to gauge the health of a storage system: high hit rates are considered a sign that a system is functioning properly, while poor hit rates suggest that tuning or configuration changes may be required. One problem with this simplistic view is that the combined hit rates of multiple independent workloads can be dominated by a single workload, thereby hiding potential problems.

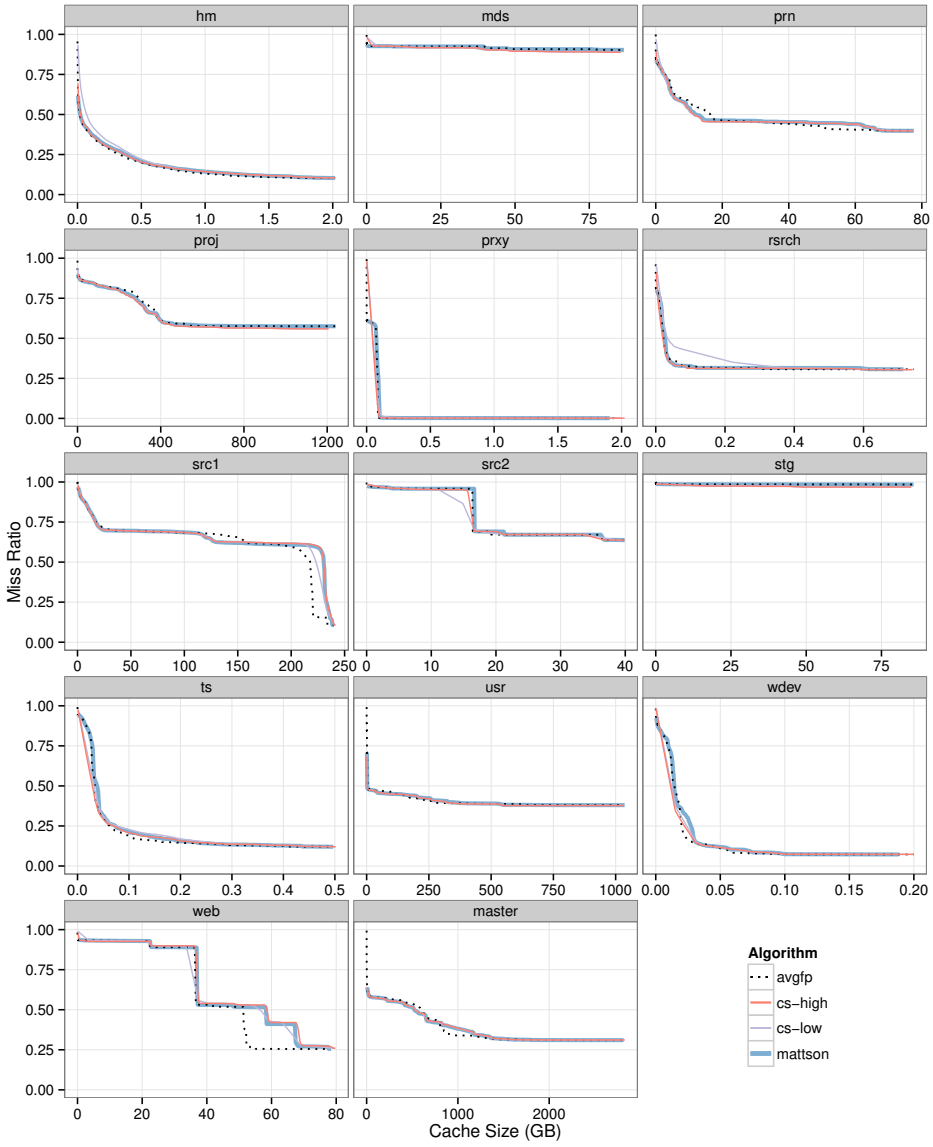


Figure 4.4: MSR miss ratio curves

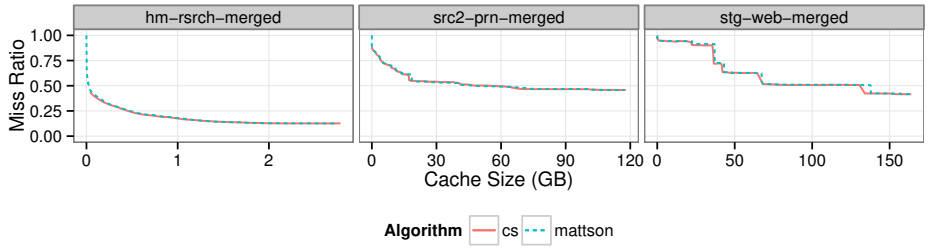


Figure 4.5: MRCs for various combinations of MSR workloads (produced by the *join* operation)

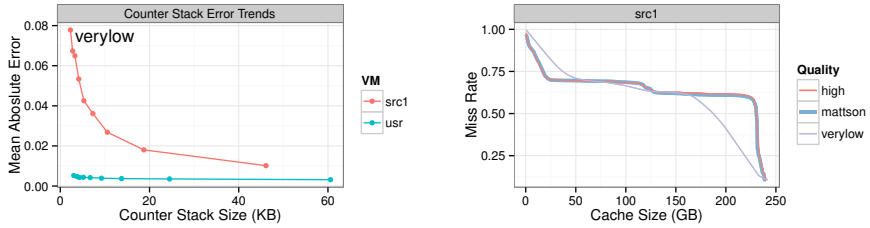


Figure 4.6: The qualitative effect of counter stack fidelity is workload-dependent. On the left, we show the curve error and file sizes of different fidelities. The `usr` workload is robust to compression to very low fidelity, while the `src1` workload degrades progressively. On the right, we show the visual outcome of compression to both *high* and *verylow* fidelity on `src1`.

We find this is indeed the case for the MSR traces. The `prxy` workload features a small working set and a high activity rate – it accesses only 2 GB of unique data over the entire week but issues 15% of all read requests in the combined trace. Table 4.2 puts this in perspective: the combined workload achieves a hit rate of 50% with a 550 GB cache; more than 250 GB of additional cache capacity would be required to achieve this same hit rate without the `prxy` workload. This illustrates why *combined hit rate is not an adequate metric of system behavior*. Diagnostic tools which present hit rates as an indicator of storage well-being should be careful to consider workloads independently as well as in combination.

Desired Hit Rate	Required Cache Size	
	With `prxy`	Without `prxy`
30%	2.5 GB	21.6 GB
40%	19.2 GB	525.5 GB
50%	566.6 GB	816.0 GB

Table 4.2: Cache sizes required to obtain desired hit rates for combined MSR workloads with and without `prxy`

4.8.2 Erratic Workloads

MRCs can be very sensitive to anomalous events. A one-off bulk read in the middle of an otherwise cache-friendly workload can produce an MRC with high miss rates, arguably mischaracterizing the workload. We wrote a simple script that identifies erratic workloads by searching for hour-long slices with unusually high miss ratios. The script found several workloads, including `mds`, `stg`, `ts`, and `prn`, whose week-long MRCs are dominated by just a few hours of intense activity.

Figure 4.7 shows the effect these bursts can have on workload performance. The full-week MRC for `prn` (Figure 4.4) shows a maximum achievable hit rate of 60% at a cache size of 83 GB. The workload features a two-hour read burst starting 102 hours into the trace which accounts for 29% of the total requests and 69% of the unique blocks. Time-sliced MRCs before and after this burst feature hit rates of 60% at cache sizes of 10 GB and 12 GB, respectively. This is a clear example of how *anomalous events can significantly distort* MRC*s*, and it shows why it is important to consider MRCs over various intervals in time, especially for long-lived workloads.

4.8.3 Conflicting Workloads

Many real-world workloads exhibit pronounced diurnal patterns: interactive workloads typically reflect natural trends in business hours, while automatic workloads are often scheduled at regular intervals throughout the day [43, 72, 101]. When

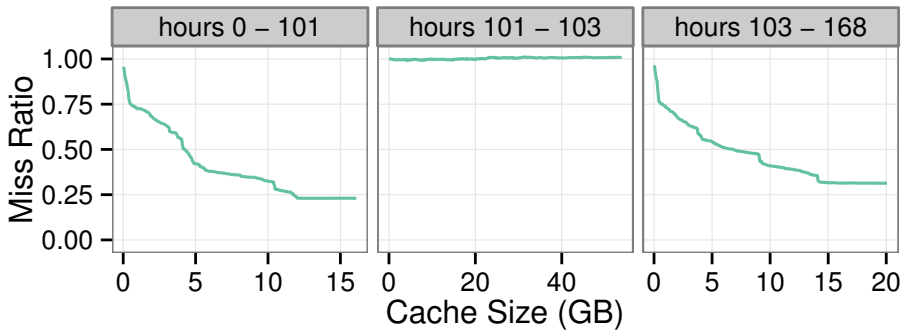


Figure 4.7: Time-sliced `prn` workload

such workloads are served by the same shared storage, it makes sense to try to limit the degree to which they interfere with one another.

The time-shifting functionality of counter stacks provides a powerful tool for exploring coarse-grain scheduling of workloads. To demonstrate this, we wrote a script which computes the MRCs of the combined MSR trace (excluding `prxy`) in which the start times of a few of the larger workloads (`proj`, `src1`, and `usr`) are shifted by up to six hours. Figure 4.8 plots the best and worst MRCs computed by this script. As is evident, *workload scheduling can significantly affect hit rates*. In this case, shifting workloads by just a few hours changes the capacity needed for a 50% hit rate by almost 50%.

4.8.4 Periodic Workloads

MRCs are good at characterizing the raw capacity needed to accommodate a given working set, but they provide very little information about how that capacity is used over time. In environments where many workloads share a common cache, this lack of temporal information can be problematic. For example, as Figure 4.4 shows, the entire working set of `web` is less than 80 GB, and it exhibits a hit rate of 75% with a dedicated cache at this size. However, as shown in Figure 4.9, the workload is highly periodic and is idle for all but a few hours every day.

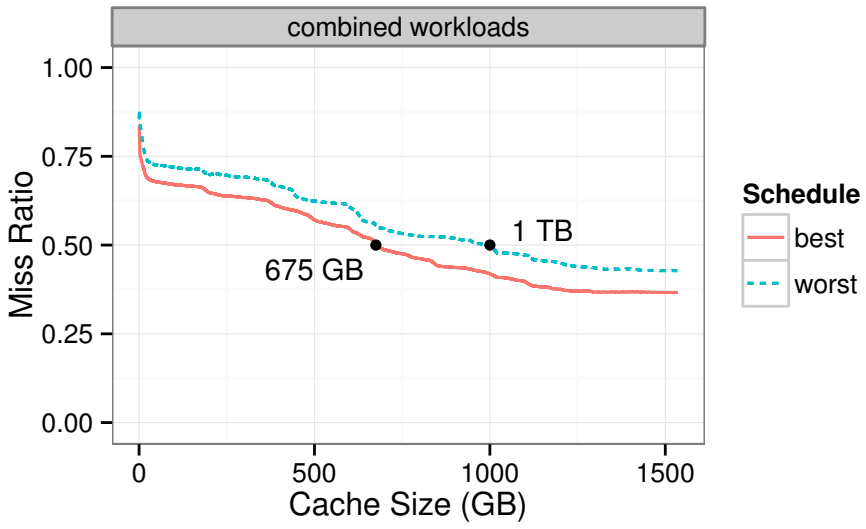


Figure 4.8: Best and worst time-shifted MRCs for MSR workloads (excluding `prxy`). We omit cache sizes greater than 1.5 TB to preserve details in the plot.

This behavior is characteristic of automated tasks like nightly backups and indexing jobs, and it can be problematic because *periodic workloads with long reuse distances tend to perform poorly in shared caches*. The cost of this is twofold: first, the periodic workloads exhibit low hit rates because their long reuse distances give them low priority in LRU caches; and second, they can penalize other workloads by repeatedly displacing 'hotter' data. This is exactly what happens to `web` in a cache shared with the rest of the MSR workloads: despite its modest working set size and high locality, it achieves a hit rate of just 7.5% in a 250 GB cache and 20% in a 500 GB cache.

Scan-resistant replacement policies like ARC [79] and CAR [17] offer one defense against this poor behavior by limiting the cache churn induced by periodic workloads. But a better approach might be to the exploit the highly regular nature of such workloads – assuming they can be identified – through intelligent prefetching. Counter stacks are well-suited for this task because they make it easy to detect periodic accesses to non-unique data. While this alone would not be sufficient to implement intelligent prefetching (because the counters do not indicate *which*

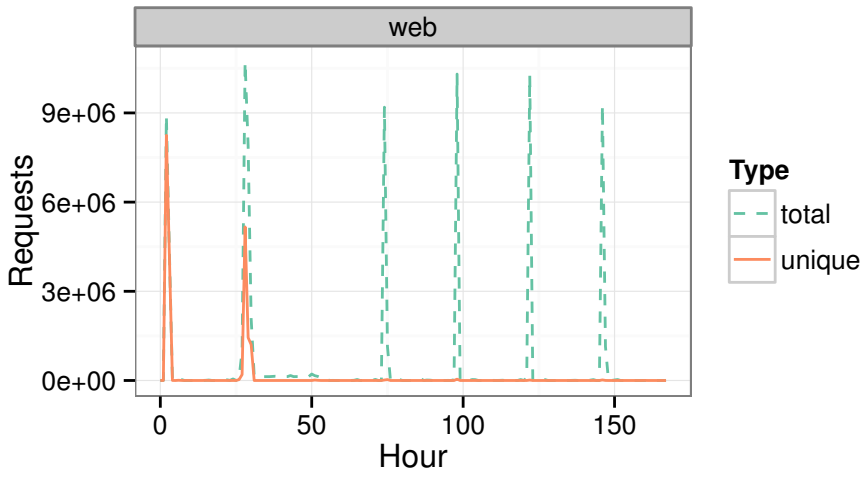


Figure 4.9: `web` total and unique requests per hour

blocks should be prefetched), it could be used to alert the system of the recurring pattern and initiate the capture of a more detailed trace for subsequent analysis.

4.8.5 Zipfian Workloads

We end with a brief discussion of synthetic workload generators like FIO [14] and IOMeter [105]. These tools are commonly used to test and validate storage systems. They are capable of generating IO workloads based on parameters describing, among other things, read/write mix, queue depth, request size, and sequentiality. The simpler among them support various combinations of random and sequential patterns; FIO recently added support for pareto and zipfian distributions, with the goal of better approximating real-world workloads.

Moving from uniform to zipfian distributions is a step in the right direction. Indeed, many of the MSR workloads, including `hm`, `mds`, and `prn`, exhibit roughly zipfian distributions. However, as is evident in Figure 4.4, the MRCs of these workloads vary dramatically. Figure 4.10 plots the MRC of a perfectly zipfian workload produced by FIO alongside two permutations of the same workload; as expected,

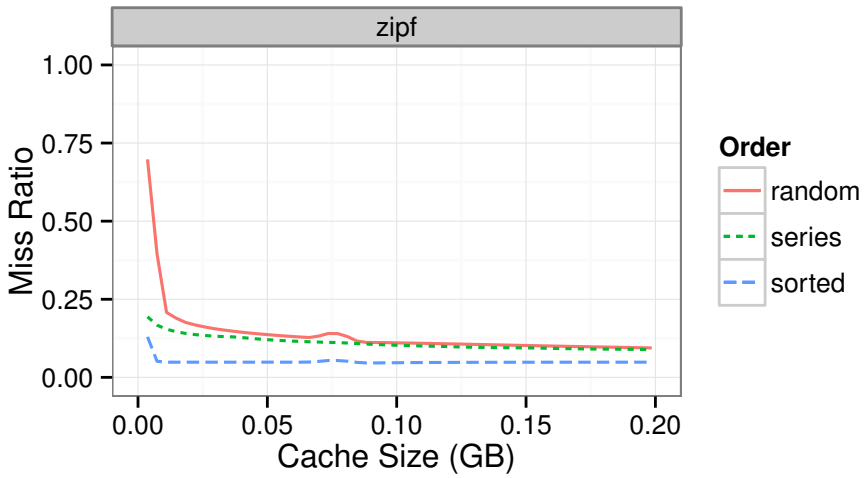


Figure 4.10: MRCs for three permutations of a single zipfian distribution: `random`, `series` (a concatenation of sorted series of unique requests), and `sorted` (truncated to preserve detail).

request ordering has a significant impact on locality and cache behavior. These figures show that *synthetic zipfian workloads do not necessarily produce 'realistic'* MRC*s*, emphasizing the importance of using real-world workloads when evaluating storage performance.

4.9 Related Work

Mattson et al. [78] defined stack distances and presented a simple $O(NM)$ time, $O(M)$ space algorithm to calculate them. Bennett and Kruskal [20] used a tree-based implementation to bring the runtime to $O(N\log(N))$. Almási et al. improved this to $O(N\log(M))$, and Niu et al. [89] introduced a parallel algorithm.

A different line of work explores techniques to efficiently approximate stack distances. Eklov and Hagersten [42] proposed a method to estimate stack distances based on sampling. Ding and Zhong [38] use an approximation technique inspired by the tree-based algorithms. Xiang et al. [125] define the footprint of a given trace window to be the number of distinct blocks occurring in the window. Using reuse

distances, they estimate the average footprint across a logarithmic scale of window lengths. Xiang et al. [126] then develop a theory connecting the average footprint and the miss ratio, contingent on a regularity condition they call the *reuse-window hypothesis*. In comparison, counter stacks use dramatically less memory while producing MRCs with comparable accuracy.

A large body of work from the storage community explores methods for representing workloads concisely. Chen et al. [31] use machine learning techniques to extract workload features, Tarasov et al. [111] describe workloads with feature matrices, and Delimitrou et al. [35] model workloads with Markov Chains. These representations are largely incomparable to counter stacks – they capture many details that are not preserved in counter stack streams, but they discard much of the temporal information required to compute accurate MRCs.

Many domain-specific compression techniques have been proposed to reduce the cost of storing and processing workload traces. These date back to Smith's stack deletion [106] and include Burtscher's VPC compression algorithms [24]. They generally preserve more information than counter stacks but achieve lower compression ratios. They do not offer new techniques for MRC computation.

4.10 Conclusion

Sizing the tiers of a hierarchical memory system and managing data placment across them is a difficult, workload dependent problem. Techniques such as miss ratio curve estimation have existed for decades as a method of modeling workload behaviors offline, but their computational and memory overheads have prevented their incorporation as a means to make live decisions in real systems. Even as an offline tool, practical issues such as the overheads associated with trace collection and storage often prevent the sharing and analysis of memory access traces.

Counter stacks provide a powerful software tool to address these issues. They are a compact form of locality characterization that allow workloads to be studied in new interactive ways, for instance by searching for anomalies or shifting workloads to identify pathological load possibilities. They can also be incorporated directly

into system design as a means of making more informed and workload-specific decisions about resource allocation across multiple tenants.

While the design and implementation of counter stacks described in this chapter have been motivated through the design of an enterprise storage system, the techniques are relevant in other domains, such as processor architecture, where the analysis of working set size over time and across workloads is critical to the design of efficient, high-performance systems.

Chapter 5

Mirador: An Active Control Plane for Datacenter Storage

A version of this chapter was published at the 15th USENIX Conference on File and Storage Technologies in 2017 [120].

5.1 Introduction

In becoming an active resource within the datacenter, storage is now similar to the compute and network resources to which it attaches. For those resources, recent years have seen a reorganization of software stacks to cleanly disentangle the notions of control and data paths. This thrust toward "software defined" systems aims for designs in which virtualized resources may be provisioned on demand and in which central control logic allows the programmatic management of resource placement in support of scale, efficiency, and performance.

This chapter observes that modern storage systems both warrant and demand exactly this approach to design. The emergence of high-performance rack-scale hardware [13, 44, 92] is amplifying the importance of *connectivity* between application workloads and their data as a critical aspect of efficient datacenter design. Fortu-

nately, the resource programmability introduced by software defined networks and the low cost of data migration on non-volatile memory means that the dynamic reconfiguration of a storage system is achievable.

How is dynamic placement useful in the context of storage? First, consider that network topology has become a very significant factor in distributed storage designs. Driven by the fact that intra-rack bandwidth continues to outpace east/west links and that storage device latencies are approaching that of Ethernet round-trip times, efficient storage placement should ensure that data is placed in the same rack as the workloads that access it, and that network load is actively balanced across physical links.

A separate goal of distributing replicas across isolated failure domains requires a similar understanding of physical and network topology, but may act in opposition to the goal of performance and efficiency mentioned above. While placement goals such as these examples can be motivated and described in relatively simple terms, the resulting placement problem is multi-dimensional and continuously changing, and so very challenging to solve.

Mirador is a dynamic storage placement service that addresses exactly this problem. Built as a component within a scale-out enterprise storage product [34], Mirador's role is to translate configuration *intention* as specified by a set of *objective functions* into appropriate placement decisions that continuously optimize for performance, efficiency, and safety. The broader storage system that Mirador controls is capable of dynamically migrating both the placement of individual chunks of data and the client network connections that are used to access them. Mirador borrows techniques from dynamic constraint satisfaction to allow multi-dimensional goals to be expressed and satisfied dynamically in response to evolutions in environment, scale, and workloads.

This chapter describes our experience in designing and building Mirador, which is the second full version of a placement service we have built. Our contributions are threefold: We demonstrate that robust placement policies can be defined as simple declarative objective functions and that general-purpose solvers can be used to find solutions that apply these constraints to both network traffic and data place-

ment in a production storage system, advancing the application of optimization techniques to the storage configuration problem [1, 8, 10, 11, 110]. We show that for performance-dense storage clusters, placement decisions informed by the relative capabilities of network and storage tiers can yield improvements over more static layouts originally developed for large collections of disks. And finally, we investigate techniques for exploiting longitudinal workload profiling to craft custom placement policies that lead to additional improvements in performance and cost-efficiency.

5.2 A Control Plane for Datacenter Storage

Mirador implements the control plane of a scale-out enterprise storage system which presents network-attached block devices for use by virtual machines (VMs), much like Amazon's Elastic Block Store [19]. A typical deployment consists of one or more independent storage nodes populated with performance-dense NVMe devices, each capable of sustaining random-access throughputs of hundreds of thousands of IOPS. In order to capitalize on the low latency of these devices, storage nodes are commonly embedded horizontally throughout the datacenter alongside the compute nodes they serve. In this environment, Mirador's role is to provide a centralized placement service that continuously monitors the storage system and coordinates the migration of both data and network connections in response to workload and environmental changes.

A guiding design principle of Mirador is that placement decisions should be ***dynamic*** and ***flexible***.

Dynamic placement decisions allow the system to adapt to environmental change. We regularly observe deployments of hundreds to thousands of VMs where only a small number of workloads dominate resource consumption across the cluster at any given time. Moreover, the membership of this set often changes as VMs are created and deleted or they transition through different workload phases. For these reasons, the initial choices made when placing data in the cluster may not always be the best ones; significant improvements can often be had by periodically

re-evaluating placement decisions over time in response to changes in workload behavior.

Flexible placement decisions allow the system to articulate complex and multidimensional policy. Rather than trying to combine diverse and often conflicting goals in a single monolithic description, Mirador approaches system configuration as a search problem. Policies are composed of one or more *objective functions*, simple rules that express how resources should be allocated by computing numerical costs for specific configurations. A planning engine employs established constraint satisfaction techniques to efficiently search the configuration space for a minimal-cost solution.

In our experience, policies expressed as simple independent rules are substantially more perspicuous and robust than their monolithic alternatives. For example, after upgrading the customized planning engine that shipped in an early version of the product to a generic constraint solver, we were able to replace a load balancing policy originally defined in 2,000 lines of imperative Python with a similar policy composed of seven simple rules each expressed in less than thirty lines of code (see § 5.3.2 for examples). Much of the complexity of the original policy came from describing *how* it should be realized rather than *what* it intended to achieve. By disentangling these two questions and answering the former with a generic search algorithm, we arrived at a policy description that is equally efficient as the first version, yet much easier to reason about and maintain.

Mirador implements the configuration changes recommended by the planning engine by coordinating a cluster-wide schedule of data and network migration tasks, taking care to minimize the performance impact on client workloads. It communicates directly with switches and storage nodes to effect these migrations, continually monitoring system performance as it does so. In this way it actively responds to environmental and workload changes and results in a more responsive, robust system.

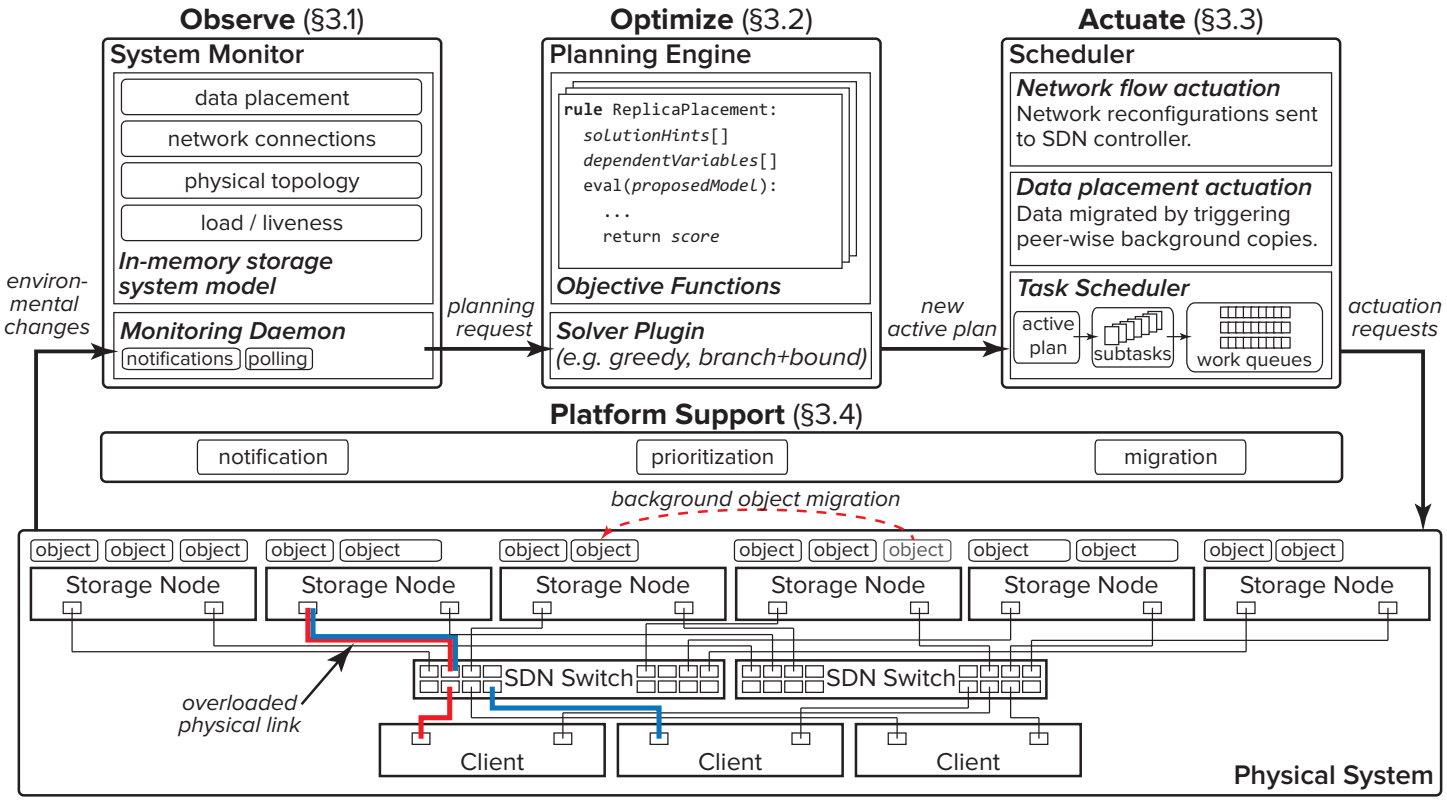


Figure 5.1: The storage system architecture (below) and the Mirador rebalance pipeline (above). The figure shows two examples of the system performing actuations in response to observed state. First, the fourth storage node has become disproportionately full relative to the other nodes. To balance capacity in the system, the rightmost object on that node is undergoing background migration to the third node. Second, the physical network link into the left side port of the second storage node has come under pressure from two high-volume flows from the first two clients. The system will observe this overload, and then chose one of the flows to migrate to a different physical link.

5.3 Mirador

Mirador is a highly-available data placement service that is part of a commercial scale-out storage product. Figure 5.1 presents a typical cluster composed of multiple storage *nodes*. Each node is a regular server populated with one or more directly-attached, non-volatile storage *devices*. Nodes implement an object interface on top of these devices and manage virtual to physical address translations internally. Objects present sparse 63-bit address spaces and are the primary unit of placement. A virtual block device interface is presented to clients. Virtual devices may be composed of one or more objects distributed across multiple nodes; by default, they are striped across 16 objects, resulting in typical object sizes on the order of tens to hundreds of GiB.

The storage cluster is fronted by a set of Software Defined Network (SDN) switches that export the cluster over a single virtual IP address. Clients connect to the virtual IP and are directed to storage nodes by a custom SDN controller. Nodes are connected in a mesh topology, and any node is capable of servicing requests from any client, allowing the mapping between clients and nodes to be modified arbitrarily.

One or more nodes in the cluster participate as a Mirador service provider. Service providers work together to monitor the state of the cluster and initiate *rebalance jobs* in response to topology and load changes. Rebalance jobs are structured as a control pipeline that generates and executes plans for dynamically reconfiguring the placement of data and client connections in order to optimize for performance, efficiency, and safety. Job state is periodically checkpointed in a replicated state machine [59], providing strong resliency against failures.

The rebalance pipeline is composed of three stages:

Observation A *system monitor* collects resource metrics like device and network load along with detailed workload profiles to construct a model of the cluster.

Optimization A *planning engine* computes a numerical cost for the current configuration and searches for alternative configurations that would reduce or eliminate this cost. If a lower-cost arrangement is identified, a plan is constructed that yields the desired results.

Actuation A *scheduler* implements the plan by coordinating the migration of data and client connections.

5.3.1 Observation

The system monitor maintains a *storage system model* that captures all relevant properties of the physical system, including static features like cluster topology (e.g., the number of devices and nodes, the capacity of their network links, and user-defined failure domains) and dynamic features like the current free space and IO load of devices and the utilization of network ports.

The monitor also collects highly-compressed sketches of individual workload behavior [122]. These summaries are collected by a dedicated *workload analysis* service, and they include features such as *miss ratio curves* and *windowed footprints*. Unlike hardware utilization levels, this data cannot be computed from instantaneous measurements, but instead requires detailed profiling of workloads over extended periods of time.

The monitor synchronizes the model by polling the cluster; sampling frequencies vary from every few seconds for metrics like link load to tens of minutes for workload footprint measurements, while exceptional events such as device failures are signalled via special alerts.

5.3.2 Optimization

The planning engine implements the logic responsible for generating rebalance plans. Placement logic is encapsulated in one or more *objective functions* that specify rules for how data and flows should be distributed across the cluster. The

engine invokes a *solver* to search for new configurations that reduce placement costs, as defined by the objective functions.

The planning engine manipulates a copy of the storage model when considering alternative configurations. For example, if a decision is made to move an object from one device to another, the modelled free space and load of each device is adjusted to reflect the change.

Modelling data migration within the cluster is a challenging problem. While an object's size serves as a rough approximation of the cost of migrating it, the actual time required to move the data depends on many things, including the type and load of the source and destination devices, network contention along the migration path, and fragmentation of the data being migrated. This is important, however, because system resources like free space and bandwidth may be consumed at both the source and destination devices during migration, and the solver may make poor decisions if this usage is modelled incorrectly. For this reason, migrations initiated during the optimization stage are modelled conservatively by *reserving* space on the destination device at the beginning of operation and only releasing it from the source device once the migration has completed.

Objective Functions

Data placement is expressed as an optimization problem by representing objects and flows as variables and devices and links as the values these variables can take, respectively. Within this framework, objective functions model the cost (or benefit) of assigning a value to a given variable (e.g., placing a replica on a specific device). [1]

Mirador objective functions can assign arbitrary numerical costs to a given configuration. Hard constraints, implemented by rules imposing an infinite cost, can never be violated – any configuration with an infinite cost is rejected outright. Negative costs can also be used to express *affinities* for preferred assignments. An *optimal*

[1] For clarity of exposition, we use the terms *objective function* and *rule* interchangably throughout the chapter.

configuration is one that minimizes the cumulative cost of all assignments; solvers employ various search strategies to find minimal-cost solutions. In the case that no finite-cost configuration can be found (e.g., due to catastrophic hardware failure), Mirador raises an alert that manual intervention is required.

Objective functions are expressed as simple Python functions operating on the storage system model described above. Listing 5.1 shows a rule designed to minimize load imbalances by stipulating that the spread between the most- and least-loaded devices falls within a given range. (Note that this formulation codifies a system-level notion of balance by assigning costs to *all* objects located on overloaded devices; moving just one such object to a different device may be enough to eliminate the cost for all the remaining objects.) During the optimization stage, the planning engine converts the storage model into an abstract representation of variables, values, and objectives, and computes the cost of each assignment by invoking its associated rules (see § 5.3.2).

A special annotation specifies the *scope* of the rule, indicating which components it affects (e.g., objects, connections, devices, links). Solvers refer to these annotations when determining which rules need to be re-evaluated during configuration changes. For example, the `load_balanced` rule affects `devices`, and must be invoked whenever the contents of a device changes.

Mutual objectives can be defined over multiple related objects. For instance, Listing 5.2 gives the implementation of a rule stipulating that no two objects in a replica set reside on the same device; it could easily be extended to include broader knowledge of rack and warehouse topology as well. Whenever a solver assigns a new value to a variable affected by a mutual objective, it must also re-evaluate all related variables (e.g., all other replicas in the replica set), as their costs may have changed as a consequence of the reassignment.

Rules can provide hints to the solver to help prune the search space. Rule implementations accept a *domain* argument, which gives a dictionary of the values that can be assigned to the variable under consideration, and is initially empty. Rules are free to update this dictionary with the expected cost that would be incurred by assigning a particular value. For example, the rule in Listing 5.2 populates a given

replica's domain with the pre-computed cost of moving it onto *any* device already hosting one of its copies, thereby deprioritizing these devices during the search. The intuition behind this optimization is that most rules in the system only affect a small subset of the possible values a variable can take, and consequently, a handful of carefully chosen hints can efficiently prune a large portion of the solution space.

A policy consists of one or more rules, which can be restricted to specific hardware components or object groups in support of multi-tenant deployments.

```
@rule(model.Device)
def load_balanced(fs, device, domain):
   cost, penalty = 0, DEVICE_BALANCED_COST
   # compute load of current device
   # for the current sample interval
   load = device.load()
   # compute load of least-loaded device
   minload = fs.mindevice().load()
   if load - minload > LOAD_SPREAD:
      # if the difference is too large,
      # the current device is overloaded
      cost = penalty
   return cost
```

Listing 5.1: Load balancing rule

```
@rule(model.ReplicaSet)
def rplset_devices_unique(fs, replica, domain):
   cost, penalty = 0, INFINITY
   for rpl in replica.rplset:
      if rpl is replica:
         # skip current replica
         continue
      if rpl.device is replica.device:
         # two replicas on the same device
         # violate redundancy constraint
         cost = penalty
      # provide a hint to the solver that the
      # devices already hosting this replica set
      # are poor candidates for this replica.
      domain[rpl.device] += penalty
   return cost
```

Listing 5.2: Hardware redundancy rule

Solvers

The planning engine is written in a modular way, making it easy to implement multiple solvers with different search strategies. Solvers accept three arguments: a dictionary of *assignments* mapping variables to their current values, a dictionary of *domains* mapping variables to all possible values they can take, and a dictionary of *objectives* mapping variables to the rules they must satisfy. Newly-added variables may have no assignment to start with, indicating that they have not yet been placed in the system. Solvers generate a sequence of *solutions*, dictionaries mapping variables to their new values. The planning engine iterates through this sequence of solutions until it finds one with an acceptable cost, or no more solutions can be found.

Mirador provides a pluggable solver interface that abstracts all knowledge of the storage model described abover. Solvers implement generic search algorithms and are free to employ standard optimization techniques like forward checking [54] and constraint propagation [77] to improve performance and solution quality.

We initially experimented with a branch and bound solver [97] because at first glance it fits well with our typical use case of soft constraints in a dense solution space [48]. A key challenge to using backtracking algorithms for data placement, however, is that these algorithms frequently yield solutions that are very different from their initial assignments. Because reassigning variables in this context may imply migrating a large amount of data from one device to another, this property can be quite onerous in practice. One way to address this is to add a rule whose cost is proportional to the difference between the solution and its initial assignment (as measured, for example, by its Hamming distance) [55]. However, this technique precludes zero-cost reconfigurations (since every reassignment incurs a cost) and thus requires careful tuning when determining whether a solution with an acceptable cost has been found.

We eventually adopted a simpler greedy algorithm. While it is not guaranteed to identify optimal solutions in every case, we find in practice that it yields quality solutions with fewer reassignments and a much more predictable run time. In fact,

the greedy algorithm has been shown to be a 2-approximate solution for the related makespan problem [52], and it is a natural fit for load rebalancing as well [3].

Listing 5.3 presents a simplified implementation of the greedy solver. It maintains a priority queue of variables that are currently violating rules, ordered by the cost of the violations, and a priority-ordered domain for each variable specifying the possible values it can take. A pluggable module updates domain priorities in response to variable reassignments, making it possible to model capacity and load changes as the solver permutes the system searching for a solution. The current implementation prioritizes values according to various utilization metrics, including free space and load.

As described in § 5.3.2, objective functions can provide hints to the solver about potential assignments. The greedy algorithm uses these hints to augment the priority order defined by the storage system model, so that values that would violate rules are deprioritized. The search is performed in a single pass over all variables, starting with the highest-cost variables. First the rules for the variable are invoked to determine whether any values in its domain violate the prescribed placement objectives (or alternatively, satisfy placement affinities). If the rules identify a zero or negative-cost assignment, this is chosen. Otherwise, the highest-priority unconstrained value is selected from the variable's domain. The search yields its solution once all violations have been resolved or all variables have been evaluated.

Besides its predictable run time, the greedy algorithm generally yields low migration overheads, since only variables that are violating rules are considered for reassignment. However, if the initial assignments are poor, the algorithm can get trapped in local minima and fail to find a zero-cost solution. In this case, a second pass clears the assignment of a group of the costliest variables collectively, providing more freedom for the solver, but potentially incurring higher migration costs. We find that this second pass is rarely necessary given the typically underconstrained policies we use in production and is limited almost exclusively to unit tests that intentionally stress the planning engine (see § 5.5 for more details).

```
def greedy(assignments, domains, objectives):
  # rank variables according to cost
  queue = PriorityQueue(domains)

  while queue.cost() > 0:
    # select the highest-cost variable
    val = None
    var = queue.pop()
    cur = assignments.get(var)
    domain = domains[var]

    # retrieve the variable's current cost and any domain hints provided
    # by the rules
    cost, hints = score(var, cur, objectives)
    if cost <= 0:
      continue # current assignment is good

    if hints:
      # find the lowest-cost hint; typically, most values are
      # unconstrained, so this linear scan adds a small constant overhead
      try:
        val = min(v for v in hints if v in domain and v != cur)
      except ValueError:
        pass

    if val is None or hints[val] > 0:
      # if we have no hints, or the best hints are costly, choose the
      # lowest-cost unconstrained value in the domain
      val = next((v for v in domain if v not in hints and v != cur), val)

    if val is None:
      c = infinity # couldn't find a value
    else:
      c, _ = score(var, val, objectives) # compute cost of new value

    if c >= cost:
      continue # no benefit to re-assigning

    assignments[var] = val # we found a better assignment

    # recompute the cost of any mutually-constrained variables that
    # haven't already been evaluated
    for v in rulemap(var, objectives):
      if v in queue:
        queue.reschedule(v)

  return assignments # we've arrived at a solution
```

Listing 5.3: Greedy solver

5.3.3 Actuation

Mirador can migrate both data and client connections. The scheduler models the cost of data migration conservatively, and attempts to minimize the impact of such migrations on client performance whenever possible. Connection migrations are generally cheaper to perform and as such occur much more frequently – on the order of minutes rather than hours.

Optimally scheduling data migration tasks is NP-hard [65–67]; Mirador implements a simple global scheduler that parallelizes migrations as much as possible without overloading individual devices or links.

Data migrations are performed in two steps: first, a background task copies an object to the destination device, and then, only after the object is fully replicated at the destination, it is removed from the source. This ensures that the durability of the object is never compromised during migration. Client connections are migrated using standard SDN routing APIs augmented by custom protocol handlers that facilitate session state handover.

5.3.4 Platform Support

Mirador executes rebalance jobs in batches by (1) selecting a group of objects and/or client connections to inspect, (2) invoking the planning engine to search for alternative configurations for these entities, and (3) coordinating the migration tasks required to achieve the new layout. Batches can overlap, allowing parallelism across the three stages. Mirador attempts to prioritize the worst offenders in early batches in order to minimize actuation costs, but it guarantees that every object is processed at least once during every job.

Mirador is able to perform its job efficiently thanks to three unique features provided by the storage platform. First, the system monitor relies on a *notification* facility provided by the cluster metadata service to quickly identify objects that have been recently created or modified. This allows nodes in the cluster to make quick, conservative placement decisions on the data path while making it easy for

Name	Objective	Cost	Lines of Code
device_has_space	devices are not filled beyond capacity	∞	4
rplset_durable	replica sets are adequately replicated on healthy devices	∞	4
load_balanced	load is balanced across devices	70	13
links_balanced	load is balanced across links	20	13
node_local	client files are co-located on common nodes	60	30
direct_connect	client connections are routed directly to their most-frequently accessed nodes	10	14
wss_best_fit	active working set sizes do not exceed flash capacities	40	4
isolated	cache-unfriendly workloads are co-located	20	30
co_scheduled	competing periodic workloads are isolated	20	35

Table 5.1: Objective functions used in evaluation section; *cost* gives the penalty incurred for violating the rule.

Mirador to inspect and modify these decisions in a timely manner, providing a strong decoupling of data and control paths. Second, the planning engine makes use of a *prioritization* interface implemented at each node that accepts a metric identifier as an argument (e.g., network or disk throughput, storage IOPS or capacity) and returns a list of the busiest workloads currently being serviced by the node. Mirador can use this to inspect problematic offenders first when attempting to minimize specific objective functions (such as load balancing and capacity constraints) rather than inspecting objects in arbitrary order. Finally, the actuation scheduler implements plans with the help of a *migration* routine that performs optimized background copies of objects across nodes and supports online reconfiguration of object metadata. This interface also provides hooks to the network controller to migrate connections and session state across nodes.

Objects	Devices	Reconfigurations	Time (seconds)
1K	10	6.40 ± 2.72	0.40 ± 0.06
1K	100	145.50 ± 33.23	0.83 ± 0.08
1K	1000	220.00 ± 12.53	10.11 ± 0.49
10K	10	0.00 ± 0.00	1.61 ± 0.01
10K	100	55.70 ± 5.46	5.54 ± 0.37
10K	1000	1475.00 ± 69.70	16.71 ± 0.88
100K	10	0.00 ± 0.00	17.10 ± 0.37
100K	100	9.30 ± 4.62	22.37 ± 5.38
100K	1000	573.80 ± 22.44	77.21 ± 2.87

Table 5.2: Greedy solver runtime for various deployment sizes with a basic load-balancing policy; *reconfigurations* gives the number of changes made to yield a zero-cost solution.

5.4 Evaluation

In this section we explore both the expressive power of Mirador policies and the impact such policies can have on real storage workloads. Table 5.1 lists the rules featured in this section; some have been used in production deployments for over a year, while others are presented to demonstrate the breadth and variety of placement strategies enabled by Mirador.

§ 5.4.1 measures the performance and scalability of the planning engine, independent of storage hardware. § 5.4.2 shows how Mirador performs in representative enterprise configurations; storage nodes in this section are equipped with 12 1 TB SSDs, two 10 gigabit Ethernet ports, 64 GiB of RAM, and 2 Xeon E5-2620 processors at 2 GHz with 6 cores each and hyperthreading enabled. § 5.4.3 and § 5.4.4 highlight the flexibility of rule-based policies, as measured on a smaller development cluster where 2 800 GB Intel 910 PCIe flash cards replace the 12 SSDs on each node.

Client workloads run in virtual machines hosted on four Dell PowerEdge r420 boxes running VMware ESXi 6.0, each with two 10 gigabit Ethernet ports, 64 GiB of RAM, and 2 Xeon ES-2470 processors at 2.3 GHz with 8 cores and hyperthreading enabled. Clients connect to storage nodes using NFSv3 via a dedicated 48-port SDN-controlled Arista 7050Tx switch, and VM disk images are striped across six-

teen objects.

5.4.1 Optimization

We begin by benchmarking the greedy solver, which is used in all subsequent experiments. Given rules that run in constant time, this solver has a computational complexity of $O(N \log N \log M)$ for a system with N objects and M devices.

We measure solver runtime when enforcing a simple load-balancing policy (based on the `device_has_space` and `load_balanced` rules, with the latter enforcing a `LOAD_SPREAD` of 20%) in deployments of various sizes. In each experiment, a simulated cluster is modelled with fixed-capacity devices (no more than ten per node) randomly populated with objects whose sizes and loads are drawn from a Pareto distribution, scaled such that no single object exceeds the capacity of a device and the cluster is roughly 65% full. For each configuration we present the time required to find a zero-cost solution as well as the number of reconfigurations required to achieve the solution, averaged over ten runs. Some experiments require no reconfigurations because their high object-to-device ratios result in very small objects that yield well-balanced load distributions under the initial, uniformly random placement; the runtimes for these experiments measure only the time required to validate the initial configuration.

As Table 5.2 shows, the flexibility provided by Python-based rules comes with a downside of relatively high execution times (more than a minute for a system with 100K objects and 1K devices). While we believe there is ample opportunity to improve our unoptimized implementation, we have not yet done so, primarily because rebalance jobs run in overlapping batches, allowing optimization and actuation tasks to execute in parallel, and actuation times typically dominate.

5.4.2 Actuation

In the following experiment we measure actuation performance by demonstrating how Mirador restores redundancy in the face of hardware failures. We pro-

vision four nodes, each with 12 1 TB SSDs, for a total of 48 devices. We deploy 1,500 client VMs, each running `fio` [14] with a configuration modelled after virtual desktop workloads. VMs issue 4 KiB requests against 1 GiB disks. Requests are drawn from an 80/20 Pareto distribution with an 80:20 read:write ratio; read and write throughputs are rate-limited to 192 KiB/sec and 48 KiB/sec, respectively, with a maximum queue depth of 4, generating an aggregate throughput of roughly 100K IOPS.

Five minutes into the experiment, we take a device offline and schedule a rebalance job. The `rplset_durable` rule assigns infinite cost to objects placed on failed devices, forcing reconfigurations, while load-balancing and failure-domain rules prioritize the choice of replacement devices. The job defers actuation until a 15 minute stabilization interval expires so that transient errors do not trigger unnecessary migrations. During this time it inspects more than 118,000 objects, and it eventually rebuilds 3053 in just under 20 minutes, with negligible effect on client workloads, as seen in Figure 5.2.

5.4.3 Resource Objectives

We now shift our attention to the efficacy of specific placement rules, measuring the degree to which they can affect client performance in live systems. We first focus on resource-centric placement rules that leverage knowledge of cluster topology and client configurations to improve performance and simplify lifecycle operations.

Topology-Aware Placement

In this experiment we measure the value of topology-aware placement policies in distributed systems. We deploy four storage nodes and four clients, with each client hosting 8 VMs running a FIO workload issuing random 4 KiB reads against dedicated 2 GiB virtual disks at queue depths ranging between 1 and 32.

Figure 5.3a presents the application-perceived latency achieved under three different placement policies when VMs issue requests at a queue depth of one. The

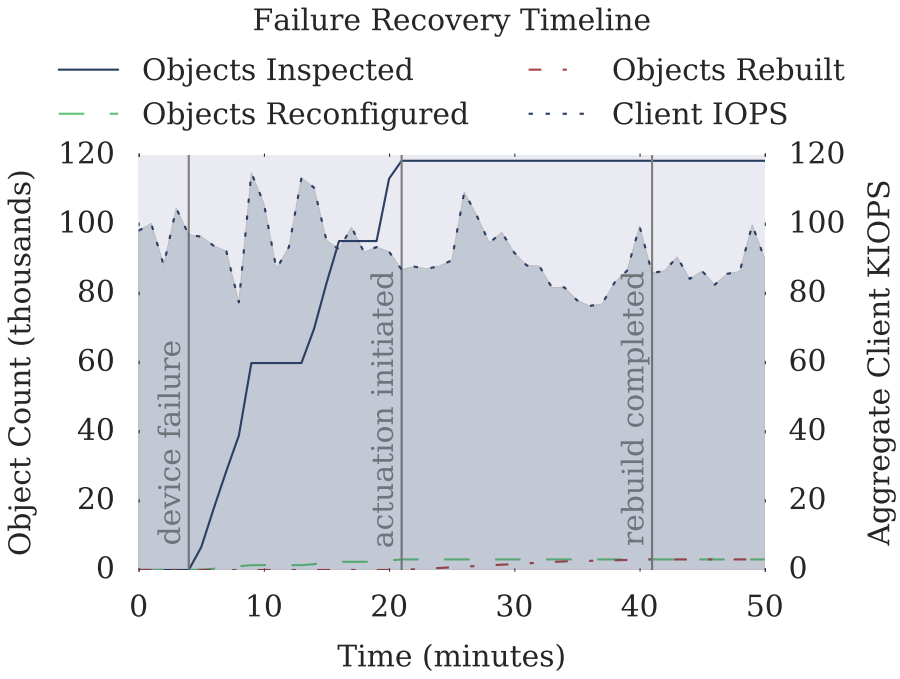
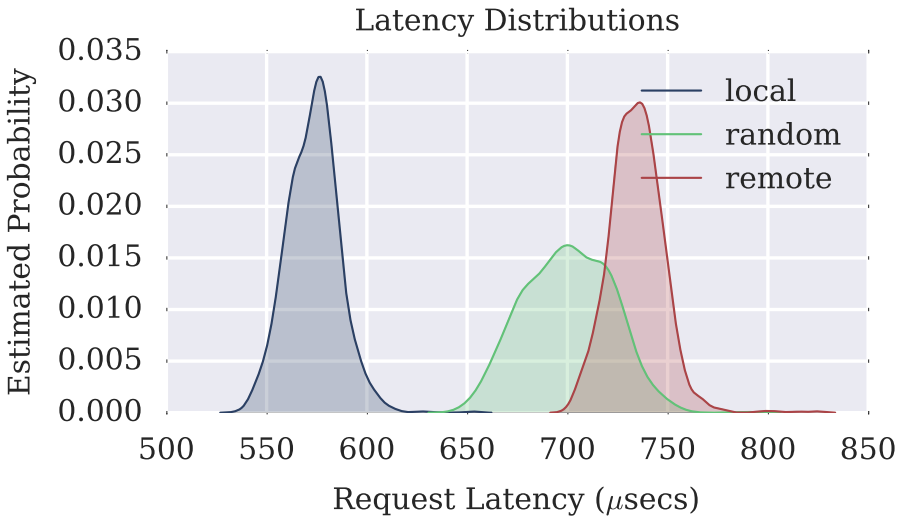
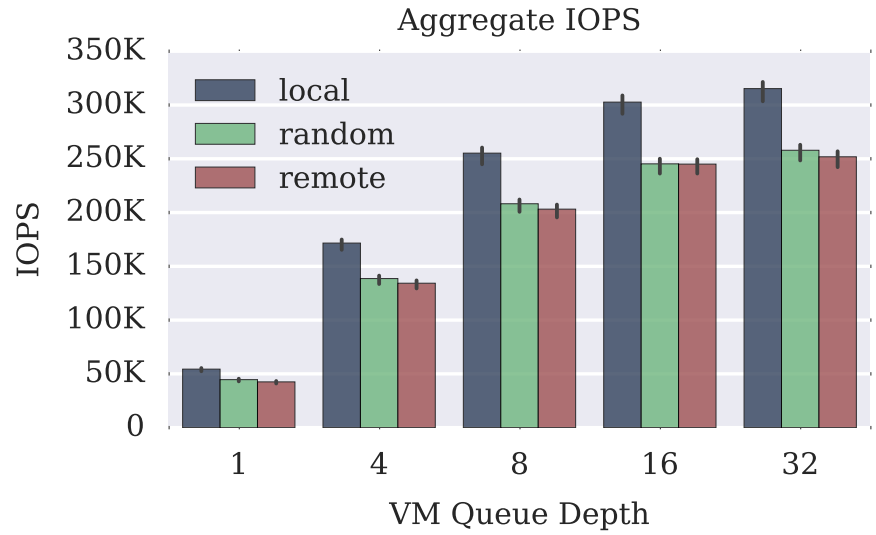


Figure 5.2: Rebuilding replicas after a device failure

random policy distributes stripes across backend devices using a simple consistent hashing scheme and applies a random one-to-one mapping from clients to storage nodes. This results in a configuration where each node serves requests from exactly one client, and with four nodes, roughly 75% of reads access remotely-hosted stripes. This topology-agnostic strategy is simple to implement, and, assuming workload uniformity, can be expected to achieve even utilization across the cluster, although it does require significant backend network communication. Indeed, as the number of storage nodes in a cluster increases, the likelihood that any node is able to serve requests locally decreases; in the limit, all requests require a backend RTT. This behavior is captured by the *remote* policy, which places stripes such that no node has a local copy of any of the data belonging to the clients it serves. The *local* policy follows the opposite strategy, placing all stripes for a given VM on a single node and ensuring that clients connect directly to the nodes hosting their



(a) Latency distributions at queue depth of 1



(b) Mean throughput at various queue depths

Figure 5.3: Performance under three different placement strategies. The *local* policy yields a median latency 18% and 22% lower than the *random* and *remote* policies, respectively, resulting in an average throughput increase of 26%. (Error bars in Figure 5.3b give 95% confidence intervals.)

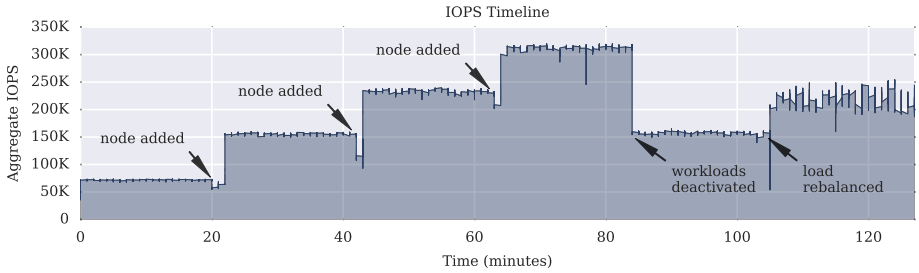


Figure 5.4: Mirador responds to changes in cluster topology and workload behavior. Data is immediately migrated to new storage nodes as they are introduced in 20 minute increments, starting at time t_{20}; the brief throughput drops are due to competition with background data copies. At time t_{85}, two of the four client machines are deactivated; the remaining client load is subsequently redistributed, at which point performance is limited by client resources.

data. Notably, all three policies are implemented in less than twenty lines of code, demonstrating the expressiveness of Mirador's optimization framework.

By co-locating VM stripes and intelligently routing client connections, the local policy eliminates additional backend RTTs and yields appreciable performance improvements, with median latencies 18% and 22% lower than those of the random and remote policies, respectively. Similar reductions are obtained across all measured queue depths, leading to comparable increases in throughput, as shown in Figure 5.3b.

Elastic Scale Out

In addition to improving application-perceived performance, minimizing cross-node communication enables linear scale out across nodes. While a random placement policy would incur proportionally more network RTTs as a cluster grows in size (potentially consuming oversubscribed cross-rack bandwidth), local placement strategies can make full use of new hardware with minimal communication overhead. This is illustrated in Figure 5.4, which presents a timeline of aggregate client IOPS as storage nodes are added to a cluster. At time t_0 the cluster is configured with a single storage node serving four clients, each hosting 16 VMs issuing

random 4 KiB reads at a queue depth of 32; performance is initially bottlenecked by the limited storage. At time t_{20}, an additional node is introduced, and the placement service automatically rebalances the data and client connections to make use of it. It takes just over two minutes to move roughly half the data in the cluster onto the new node. This migration is performed as a low-priority background task to limit interference with client IO. Two additional nodes are added at twenty minute intervals, and in each case, after a brief dip in client performance caused by competing migration traffic, throughput increases linearly.

The performance and scalability benefits of the local policy are appealing, but to be practical, this approach requires a truly dynamic placement service. While both local and random policies are susceptible to utilization imbalances caused by non-uniform workload patterns (e.g., workload 'hot spots'), the problem is exacerbated in the local case. For example, if all workloads placed on a specific node happen to become idle at the same time, that node will be underutilized. Figure 5.4 shows exactly this scenario at time t_{85}, where two clients are deactivated and the nodes serving them sit idle, halving overall throughput. After waiting for workload behavior to stabilize, the placement service responds to this imbalance by migrating some of the remaining VMs onto the idle storage, at which point the clients become the bottleneck.

5.4.4 Workload Objectives

Placement policies informed by resource monitoring can provide significant improvements in performance and efficiency, but they are somewhat *reactive* in the sense that they must constantly try to 'catch up' to changes in workload behavior. In this section we introduce and evaluate several techniques for improving data placement based on longitudinal observations of workload behavior.

The following examples are motivated by an analysis of hundreds of thousands of workload profiles collected from production deployments over the course of more than a year. The synthetic workloads evaluated here, while relatively simple, reflect some of the broad patterns we observe in these real-world profiles.

For these experiments, we extend the storage configuration described in § 5.4.3 with a disk-based capacity tier. The placement service controls how objects are assigned to flash devices as before; nodes manage the flash cards as LRU caches and page objects to disk in 512 KiB blocks. We artificially reduce the capacity of each flash device to 4 GiB to stress the tiering subsystem. While our evaluation focuses on conventional tiered storage, we note that the techniques presented here are applicable to a wide variety of hierarchical and NUMA architectures in which expensive, high-performance memories are combined with cheaper, more capacious alternatives, possibly connected by throughput-limited networks.

Footprint-Aware Placement

Many real-world workloads feature working sets (roughly defined as the set of data that is frequently accessed over a given period of time) that are much smaller than their total data sets [36, 124]. Policies that make decisions based only on knowledge of the latter may lead to suboptimal configurations. We show how augmenting traditional capacity rules with knowledge of working set sizes can lead to improved placement decisions.

We begin by deploying eight VMs across two clients connected to a cluster of two nodes. Each VM disk image holds 32 GiB, but the VMs are configured to run random 4 KiB read workloads over a fixed subset of the disks, such that working set sizes range from 500 MiB to 4 GiB. Given two nodes with 8 GiB of flash each, it is impossible to store all 256 GiB of VM data in flash; however, the total workload footprint as measured by the analysis service is roughly 17 GiB, and if carefully arranged, it can fit almost entirely in flash without exceeding the capacity of any single device by more than 1 GiB.

We measure the application-perceived latency for these VMs in two configurations. In the first, VMs are partitioned evenly among the two nodes using the *local* policy described in § 5.4.3 to avoid network RTTs. In the second, the same placement policy is used, but it is extended with one additional rule that discourages configurations where combined working set sizes exceed the capacity of a given flash card.

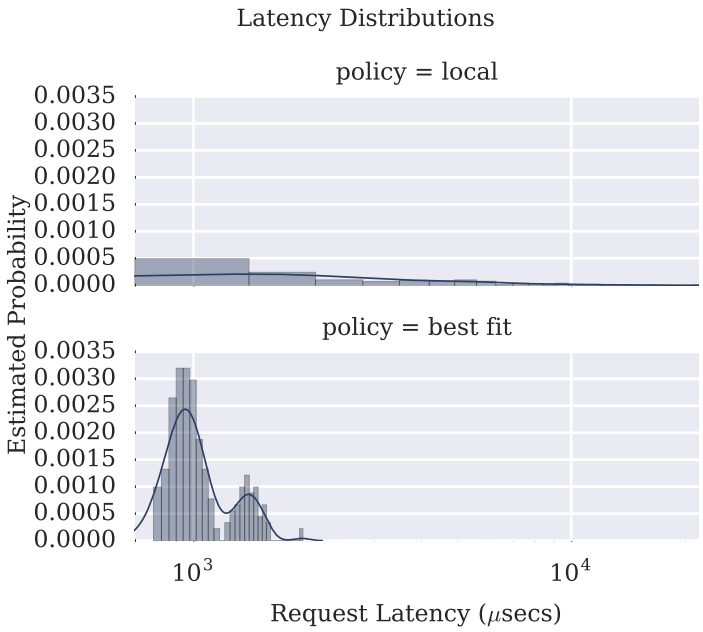


Figure 5.5: Fitting working sets to flash capacities ('best fit') yields a median latency of 997 μsecs, compared to 2088 μsecs for the 'local' policy that eliminates backend network RTTs but serves more requests from disk.

The cost of violating this rule is higher than the cost of violating the node-local rule, codifying a preference for remote flash accesses over local disk accesses. The greedy solver is a good fit for this problem and arrives at a configuration in which only one flash device serves a combined working set size larger than its capacity.

As Figure 5.5 shows, the *best-fit* policy results in significantly lower latencies, because the cost of additional network hops is dwarfed by the penalty incurred by cache misses. The purely local policy exhibits less predictable performance and a long latency tail because of cumulative queuing effects at the disk tier. This is a clear example of how combining knowledge of the relative capabilities of network links and storage tiers with detailed workload profiling can improve placement decisions.

Noisy Neighbor Isolation

We next introduce four cache-unfriendly workloads each with 4 GiB disks. The workloads perform linear scans that, given 4 GiB LRU caches, are always served from disk and result in substantial cache pollution. These workloads make it impossible to completely satisfy the working set size rule of the previous experiment.

We measure the request latency of the original workloads as they compete with these new cache-unfriendly workloads under two policies: a *fair share* policy that distributes the cache-unfriendly workloads evenly across the flash devices, and an *isolation* policy that attempts to limit overall cache pollution by introducing a new rule that encourages co-locating cache-unfriendly workloads on common nodes, regardless of whether or not they fit within flash together. As Figure 5.6 shows, this latter policy exhibits a bimodal latency distribution, with nearly 48% of requests enjoying latencies less than one millisecond while a handful of 'victim' workloads experience higher latencies due to contention with cache-unfriendly competitors. The fair share policy, on the other hand, features a more uniform distribution, with all workloads suffering equally, and a median latency more than three times higher than that of the isolated policy.

Workload Co-scheduling

Finally, we introduce a technique for leveraging long-term temporal patterns in workload behavior to improve data placement. We frequently see storage workloads with pronounced diurnal patterns of high activity at key hours of the day followed by longer periods of idleness. This behavior typically correlates with workday habits and regularly scheduled maintenance tasks [43, 87, 101]. Similar effects can be seen at much smaller scales in CPU caches, where the strategy of co-locating applications to avoid contention is called 'co-scheduling' [114].

We present a simple algorithm for reducing cache contention of periodic workloads. The workload analysis service maintains an extended time series of the footprint of each workload, where footprint is defined as the number of unique blocks accessed over some time window; in this experiment we use a window of

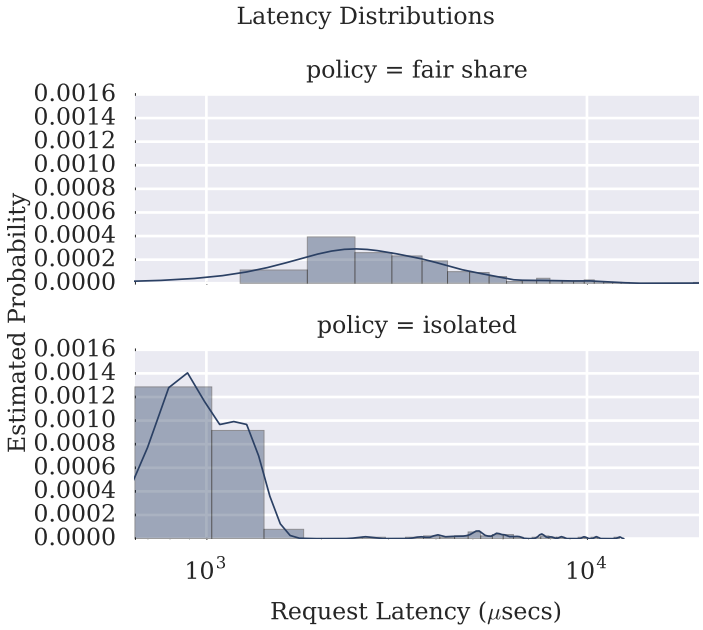


Figure 5.6: Isolating cache-unfriendly workloads on a single device yields a median latency of 1036 μsecs, compared to 3220 μsecs for the 'fair' policy that distributes these workloads uniformly across all devices.

ten minutes. Given a set of workloads, we compute the degree to which they contend by measuring how much their bursts overlap. Specifically, we model the cost of co-locating two workloads W_1 and W_2 with corresponding footprint functions $f_1(t)$ and $f_2(t)$ as $\int \min(f_1(t), f_2(t))$. We use this metric to estimate the cost of placing workloads together on a given device, and employ a linear *first-fit* algorithm [39] to search for an arrangement of workloads across available devices that minimizes the aggregate cost. Finally, we introduce the `co_scheduled` rule which encodes an affinity for assignments that match this arrangement.

We evaluate this heuristic by deploying 8 VMs with 4 GiB disks across two storage nodes each with two 4 GiB flash devices. The VMs perform IO workloads featuring periodic hour-long bursts of random reads followed by idle intervals of roughly 3 hours, with the periodic phases shifted in some VMs such that not all workloads are active at the same time. The combined footprint of any two concurrent bursts

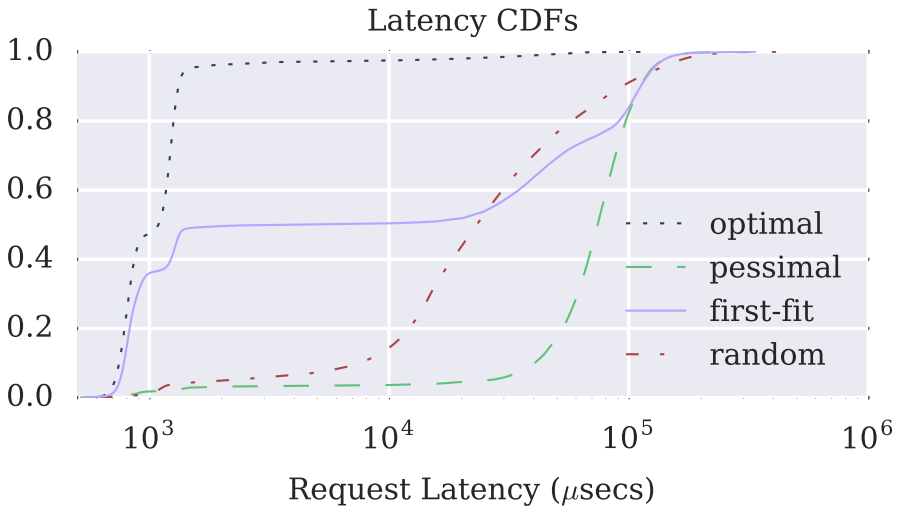


Figure 5.7: Co-scheduling periodic workloads

exceeds the size of any single flash device, and if co-located, will incur significant paging. We measure request latency under a number of different configurations: *random*, in which stripes are randomly distributed across devices, *optimal* and *pessimal*, in which VMs are distributed two to a device so as to minimize and maximize contention, respectively, and *first-fit*, as described above.

Figure 5.7 plots latency CDFs for each of these configurations. The penalty of concurrent bursts is evident from the pronounced disparity between the optimal and pessimal cases; in the latter configuration, contention among co-located workloads is high, drastically exceeding the available flash capacity. The first-fit approximation closely tracks optimal in the first two quartiles but performs more like random in the last two, suggesting room for improvement either by developing a more sophisticated search algorithm or responding more aggressively to workload changes.

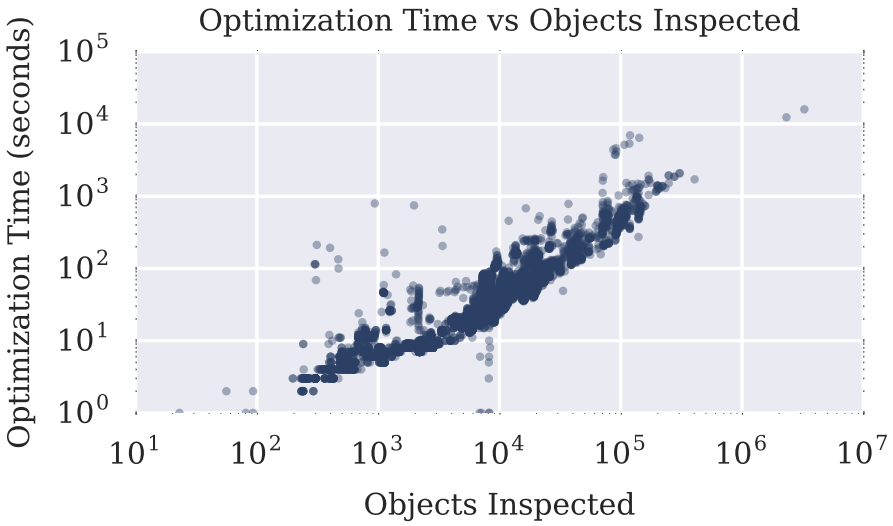


Figure 5.8: Optimization time versus objects inspected

5.5 Experience

To see how Mirador performs in real-world environments, we sample logs detailing more than 8,000 rebalance jobs in clusters installed across nearly 50 customer sites and ranging in size from 8 to 96 devices. Figure 5.8 illustrates how time spent in the optimization stage scales in proportion to the number of objects inspected; these measurements include rate-limiting delays imposed to prevent Mirador from impacting client workloads when reading metadata. Figure 5.9 plots the number of observed violations against the number of objects inspected per job, and highlights jobs that fail to find a zero-cost solution after a single optimization pass. This occurs in only 2.5% of sampled jobs in which objective functions are violated, and in 71% of these cases, no zero-cost solutions are possible due to environmental circumstances (some log samples cover periods in which devices were intentionally taken offline for testing or maintenance).

We have found Mirador's flexibility and extensibility to be two of its best attributes. Over the nearly 18 months in which it has been in production, we have adapted it

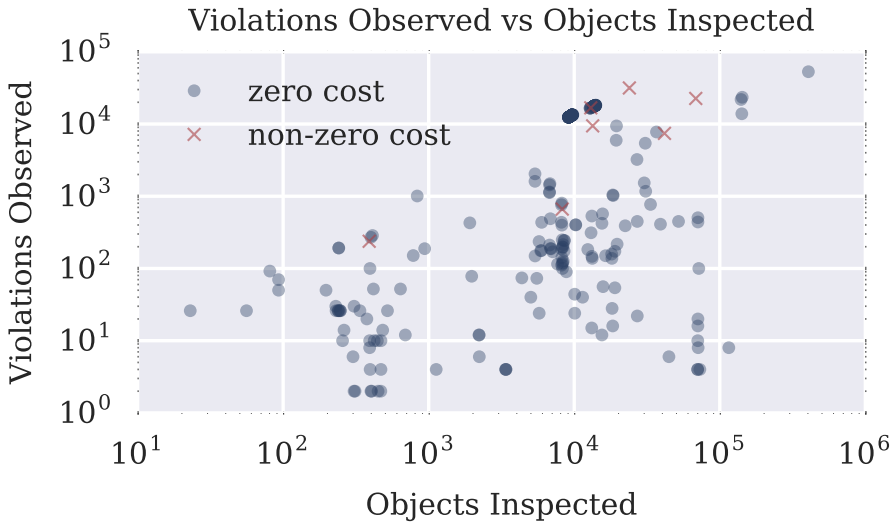


Figure 5.9: Violations observed versus objects inspected (jobs where no zero-cost solution was found after a single optimization round are marked with a red x)

to new replication policies and storage architectures simply by modifying existing rules and adding new ones. It has also been straightforward to extend Mirador to support new functionality: in addition to providing capacity balancing across storage devices and network links, it now plays a central role in cluster expansion, hardware retirement, failure recovery, health monitoring, and disk scrubbing features. For example, upon discovering an invalid data checksum, our disk scrubbing service simply marks the affected object as corrupt and notifies the placement service, where a custom rule forces the migration of marked objects to new locations, effectively rebuilding them from valid replicas in the process.

Our deployment strategy to date has been conservative: we ship a fixed set of rules (currently seven) and control how and when they are used. Assigning appropriate costs to rules requires domain knowledge, since rules often articulate conflicting objectives and poorly chosen costs can lead to unintended behavior. As an example, if solvers fail to identify a zero-cost solution, they yield the one with the lowest *aggregate* cost – if multiple rules conflict for a given assignment, the assignment

which minimizes the overall cost is chosen. It is thus important to know which objective functions a replica set may violate so that high priority rules are assigned costs sufficiently large enough to avoid priority inversion in the face of violations of multiple lower-priority rules.

While objective functions neatly encapsulate individual placement goals and are relatively easy to reason about, comprehensive policies are more complex and must be carefully vetted. We validate rules, both in isolation and combination, with hundreds of policy tests. Declarative test cases specify a cluster configuration and initial data layout along with an expected optimization plan; the test harness generates a storage system model from the specification, invokes the planning engine, and validates the output. We have also built a fuzz tester that can stress policies in unanticipated ways. The test induces a sequence of random events (such as the addition and removal of nodes, changes in load, etc.) and invokes the policy validation tool after each step. Any cluster configuration that generates a policy violation is automatically converted into a test case to be added to the regression suite after the desired behavior is determined by manual inspection. Validating *any* non-trivial placement policy can require a fair amount of experimentation, but in our experience, the cost-based framework provided by Mirador provides knobs that greatly simplify this task.

In production, rebalance jobs run in two passes: the first enforces critical rules related to redundancy and fault tolerance, while the second additionally enforces rules related to load-balancing and performance. This is done because the planning engine must inspect objects in batches (batches are limited to roughly 10,000 objects to keep memory overheads constant), and we want to avoid filling a device in an early batch in order to satisfy low-priority rules when that same device may be necessary to satisfy higher-priority rules in a later batch.

Early testing revealed the importance of carefully tuning data migration rates. Our migration service originally provided two priorities, with the higher of these intended for failure scenarios in which replicas need to be rebuilt. In practice, however, we found that such failures place additional stress on the system, often driving latencies up. Introducing high-priority migration traffic in these situations can

lead to timeouts that only make things worse, especially under load. We have since adopted a single migration priority based on an adaptive queuing algorithm that aims to isolate migration traffic as much as possible while ensuring forward progress is made.

5.6 Related Work

Researchers have proposed a wide variety of strategies for addressing the data placement problem, also known as the file assignment problem [40]. Deterministic approaches are common in large-scale systems [88, 94, 108, 115, 117] because they are decentralized and impose minimal metadata overheads, and they achieve probabilistically uniform load distribution for large numbers of objects [96, 100]. Consistent hashing [64] provides relatively stable placement even as storage targets are added and removed [51, 130]. Related schemes offer refinements like the ability to prioritize storage targets and modify replication factors [57, 58, 116], but these approaches are intrinsically less flexible than dynamic policies.

Non-deterministic strategies maintain explicit metadata in order to locate data. Some of these systems employ random or semi-random placement policies for the sake of simplicity and scalability [70, 90, 95], but others manage placement with hard-coded policies [49, 104]. Customized policies provide better control over properties such as locality and fault tolerance, which can be particularly important as clusters expand across racks [63].

Explicit metadata also make it easier to perform fine-grain migrations in response to topology and workload changes, allowing systems to redistribute load and ameliorate hot spots [73, 87]. Hierarchical Storage Management and multi-tier systems dynamically migrate data between heterogeneous devices, typically employing policies based on simple heuristics intended to move infrequently accessed data to cheaper, more capacious storage or slower, more compact encodings [4, 119].

Mirador has much in common with recent systems designed to optimize specific performance and efficiency objectives. Guerra et al. [53] describe a tiering system that makes fine-grain placement decisions to reduce energy consumption in SANs

by distributing workloads among the most power-efficient devices capable of satisfying measured performance requirements. Janus [6] is a cloud-scale system that uses an empirical cacheability metric to arrange data across heterogeneous media in a manner that maximizes reads from flash, using linear programming to compute optimal layouts. Volley [2] models latency and locality using a weighted spring analogy and makes placement suggestions for geographically distributed cloud services. Tuba [12] is a replicated key-value store designed for wide area networks that allows applications to specify latency and consistency requirements via service level agreements (SLAs). It collects hit ratios and latency measurements and periodically reconfigures replication and placement settings to maximize system utility (as defined by SLAs) while honoring client-provided constraints on properties like durability and cost. Mirador supports arbitrary cost-function optimizations using a generic framework and supports policies that control network flows as well as data placement.

Mirador also resembles resource planning systems [8, 11] like Hippodrome [10], which employ a similar observe/optimize/actuate pipeline to design cost-efficient storage systems. Given a set of workload descriptions and an inventory of available hardware, these tools search for low-cost array configurations and data layouts that satisfy performance and capacity requirements. Like Mirador, they simplify a computationally challenging multidimensional bin-packing problem by combining established optimization techniques with domain-specific heuristics. However, while these systems employ customized search algorithms with built-in heuristics, Mirador codifies heuristics as rules with varying costs and relies on generic solvers to search for low-cost solutions, making it easier to add new heuristics over time.

Ursa Minor [1] is a clustered storage system that supports dynamically configurable *m-of-n* erasure codes, extending the data placement problem along multiple new dimensions. Strunk et al. [110] describe a provisioning tool for this system that searches for code parameters and data layouts that maximize user-defined *utility* for a given set of workloads, where utility quantifies metrics such as availability, reliability, and performance. Utility functions and objective functions both provide flexibility when evaluating potential configurations; however, Mirador's greedy algorithm and support for domain-specific hints may be more appropriate for online

rebalancing than the randomized genetic algorithm proposed by Strunk et al.

5.7 Conclusion

Mirador is a placement service designed for heterogeneous distributed storage systems. It leverages the high throughput of non-volatile memories to actively migrate data in response to workload and environmental changes. It supports flexible, robust policies composed of simple objective functions that specify strategies for both data and network placement. Combining ideas from constraint satisfaction with domain-specific language bindings and APIs, it searches a high-dimension solution space for configurations that yield performance and efficiency gains over more static alternatives.

Chapter 6

Conclusion

As a commercial product, one of the features that sets Strata apart from its many competitors is the architectural support it provides for dynamic cluster reconfiguration. This is valuable for a number of reasons. First, it abolishes the much-loathed five year refresh cycle imposed by many incumbent vendors. Allowing administrators to expand clusters in response to growing demand relieves them of the burden of estimating at purchase time what their storage requirements will be many years down the road. And supporting rolling upgrades and heterogeneous clusters eliminates the need for disruptive 'forklift' upgrades in which existing systems are migrated to new hardware en masse. Second, deferring purchases until hardware is actually needed can dramatically reduce capital and operating expenses, both by allowing Moore's Law to accrue longer before money is exchanged, and by reducing the number of devices that sit idle in initially over-provisioned systems. Finally, the ability to provision performance and capacity independently gives storage administrators the flexibility they need to adapt to changing requirements within the data center.

These advantages are natural consequences of the design advocated in this thesis. The platform provided by Strata decouples logical resources from physical hardware and separates control- and data-path logic, enabling dynamic configuration changes without degrading performance, and the robust policy engine provided

by Mirador arranges for hardware resources to be allocated where they are most needed. This paradigm of abstraction, analysis, and actuation helps systems to automatically respond to changes in workload behavior and hardware configurations, a valuable capability in data center environments serving diverse workloads across large, heterogeneous clusters. It has been incredibly rewarding to see this approach succeed in real customer deployments, but it has also been instructive to observe some of its limitations. Indeed, there is still ample opportunity – and need – to continue innovating storage software, especially as hardware continues to evolve. Below I enumerate what I see as some of the most interesting directions for future improvements, some of which we have already begun to explore.

Volume Management Strata's departure from traditional aggregated designs was a response to the unprecedented performance of new PCIe flash devices like the Intel 910, which provides 800 GB of storage and serves 180,000 random read requests per second. Three years after we published the Strata paper, the Intel p3700, providing 2 TB of storage and serving 460,000 random read requests per second, hit the market at roughly the same price as the original 910. This rapid rate of progress reinforces many of the design choices we made, particularly regarding the need to efficiently virtualize hardware in support of dynamic workload multiplexing. But these new devices place even more stringent constraints on the data path: access latencies have dropped from 65 microseconds in the 910 to 20 microseconds in the p3700, and NVDIMM modules currently operate at latencies of just 10 nanoseconds. At these speeds, software overheads imposed by context switches and thread synchronization become problematic. In response, we built *Decibel*, a device virtualization layer designed to completely eliminate cross-core communication along the data path. Decibel's disaggregated architecture is similar to Strata's, but rather than presenting individual devices over the network, it presents a volume abstraction that encapsulates storage, network, *and* compute resources. Decibel volumes bind chunks of storage to dedicated cores and NIC queues; flow steering based on an explicit network addressing scheme ensures that client requests are automatically directed to the appropriate cores, eliminating the need for forwarding or synchronization in software. This, combined with a userspace net-

working stack that bypasses kernel scheduling and context switches, allows Decibel to serve remote workloads from p3700 devices at saturation with an overhead relative to local access of just 20 microseconds.

Hybrid Placement Decibel both refines and complements Strata's separation of control- and data-path logic, and it naturally benefits from the optimization techniques in Mirador that correct load imbalances and mitigate hot spots. However, while a centralized placement engine simplifies the difficult task of optimizing resource allocation, it also presents some challenges, particularly when scaling to very large deployments with billions of objects. Individually optimizing the placement of so many objects can be prohibitively expensive. Fortunately, the techniques employed by Mirador can naturally be combined with less computationally expensive approaches like statistical multiplexing to good effect. Under this regime, a deterministic policy such as consistent hashing can be used to decide the default placement of the vast majority of objects, while dynamic optimization techniques can be applied only to objects that actively contribute to performance and utilization problems. Strata's clean separation of addressing and placement facilities would naturally accommodate this hybrid approach, improving scalability without sacrificing flexibility.

Demand Swap Optimizing the placement of data in heterogeneous clusters is particularly challenging because of the huge performance variations across devices. The *demand fault* strategy conventionally used by cache replacement policies can lead to surprisingly poor performance when the combined size of active working sets is even marginally larger than the available fast storage. This technique has a tendency to penalize *many* workloads a small amount, which can become problematic as data dependencies exacerbate the effects of even a few cache misses per workload. The data we have collected from real-world deployments of production virtual machines, which comprises thousands of workload-years of detailed profiling, suggests that a better approach might be to swap entire workloads in and out of fast storage as they cycle between active and idle phases, which regularly last

hours at a time. Preliminary investigation confirms that phase changes are easily identified via counter stack analysis and that they can be predicted with fairly high confidence for a large class of workloads. We have further found that online classifiers generally identify active phases less than a minute after they begin. Given that we can reasonably expect to load entire working sets, which are typically on the order of a few dozens of gigabytes, from disk in a matter of seconds (so long as transfers are carefully scheduled across large numbers of spindles), the idea of swapping entire workloads in and out of fast storage, either reactively or speculatively, is alluring. This would leverage the sequential throughput of disks much more effectively than the demand fault approach, and would additionally make it easier to isolate ill-behaved or under-provisioned workloads.

Programmable Storage Heterogeneous clusters expose a tension between cost and performance. In many cases, purely economic constraints make this tension inevitable. However, in our experience, providing *predictable* performance is often more important than achieving device-rate speeds. Mirador uses a number of heuristics to attempt to automatically infer the optimal allocation of resources at any given time, but these heuristics do not always align with the business needs of individual customers. In situations where resources are scarce, it may be preferable to delegate allocation decisions higher up the stack, either to application developers or storage administrators. This is in keeping with recent trends in software design that have shifted traditional storage responsibilities like replication and consistency to application-level services like key/value stores and databases. These services understand the performance and placement requirements of their data better than the underlying storage system, so providing them with an interface for *safely* influencing resource allocation decisions, while protecting against buggy and malicious applications, could present new opportunities to improve performance and eliminate unwelcome surprises. Mirador's support for arbitrary soft and hard constraints provides a good starting point for this approach; exposing more of this functionality to applications would extend many of the benefits introduced by software defined networking to the storage domain.

Strata provides a solid platform for exploring these and other techniques because

of the design abstractions it provides. Implementing these abstractions in an enterprise storage product has been a labor-intensive task, but one that has yielded many benefits. In addition to producing a system that solves real problems for our customers, it has provided an opportunity to explore novel techniques for optimizing performance and efficiency within the data center, and its organizing principles offer a useful model for future system designers.

Bibliography

[1] M. Abd-El-Malek, W. V. C. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: Versatile cluster-based storage. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, FAST '05. USENIX, 2005. → pages 42, 77, 106

[2] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, and A. Wolman. Volley: Automated data placement for geo-distributed cloud services. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI '10, pages 17–32. USENIX Association, 2010. → pages 106

[3] G. Aggarwal, R. Motwani, and A. Zhu. The load rebalancing problem. *Journal of Algorithms*, 60(1):42–59, 2006. → pages 86

[4] M. K. Aguilera, K. Keeton, A. Merchant, K.-K. Muniswamy-Reddy, and M. Uysal. Improving recoverability in multi-tier storage systems. In *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 677–686. IEEE Computer Society, 2007. → pages 105

[5] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: a protoype phase change memory storage array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage '11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association. → pages 41

[6] C. Albrecht, A. Merchant, M. Stokely, M. Waliji, F. Labelle, N. Coehlo, X. Shi, and E. Schrock. Janus: Optimal flash provisioning for cloud storage

workloads. In *USENIX Annual Technical Conference*, ATC '13, pages 91–102. USENIX Association, 2013. → pages 106

[7] G. S. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 Workshop on Memory System Performance*, MSP '02, pages 37–43, 2002. → pages 47

[8] G. A. Alvarez, E. Borowsky, S. Go, T. H. Romer, R. A. Becker-Szendy, R. A. Golding, A. Merchant, M. Spasojevic, A. C. Veitch, and J. Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Transactions on Computer Systems*, 19(4):483–518, 2001. → pages 77, 106

[9] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: a fast array of wimpy nodes. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, 2009. → pages 41

[10] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. C. Veitch. Hippodrome: Running circles around storage administration. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, pages 175–188. USENIX, 2002. → pages 77, 106

[11] E. Anderson, S. Spence, R. Swaminathan, M. Kallahalla, and Q. Wang. Quickly finding near-optimal storage designs. *ACM Transactions on Computer Systems*, 23(4):337–374, 2005. → pages 77, 106

[12] M. S. Ardekani and D. B. Terry. A self-configurable geo-replicated cloud storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 367–381. USENIX Association, 2014. → pages 106

[13] K. Asanovic. Firebox: A hardware building block for 2020 warehouse-scale computers. Keynote presentation, 12th USENIX Conference on File and Storage Technologies (FAST '14), 2014. → pages 75

[14] J. Axboe. Fio–flexible I/O tester, 2011. https://github.com/axboe/fio. Visited July 2017. → pages 35, 71, 92

[15] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association. → pages 41

[16] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. Corfu: a shared log design for flash clusters. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, 2012. → pages 41

[17] S. Bansal and D. S. Modha. CAR: Clock with adaptive replacement. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, 2004. → pages 70

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, 2003. ISBN 1-58113-757-5. → pages 40

[19] J. Barr. Amazon EBS (elastic block store) - bring us your data, August 2008. https://aws.amazon.com/blogs/aws/amazon-elastic. Visited July 2017. → pages 77

[20] B. T. Bennett and V. J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975. → pages 50, 72

[21] M. Blaze. NFS tracing by passive network monitoring. In *Proceedings of the USENIX Winter 1992 Technical Conference*, pages 333–343, 1992. → pages 46

[22] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. → pages 55

[23] A. D. Brunelle. Block I/O Layer Tracing: blktrace. *HP, Gelato-Cupertino, CA, USA*, 2006. → pages 46

[24] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam. The vpc trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, 2005. → pages 73

[25] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 143–157, 2011. → pages 42

[26] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. Sane: a protection architecture for enterprise networks. In *Proceedings of the 15th USENIX Security Symposium*, SS '06, Berkeley, CA, USA, 2006. USENIX Association. → pages 26

[27] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. Mckeown, and S. Shenker. Ethane: Taking control of the enterprise. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2007. → pages 26

[28] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, 2010. → pages 41

[29] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, 2012. → pages 41

[30] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2): 4:1–4:26, June 2008. → pages 42

[31] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design implications for enterprise storage systems via multi-dimensional trace analysis. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, pages 43–56. ACM, 2011. → pages 73

[32] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. Nv-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM. → pages 42

[33] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better i/o through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems*

Principles, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM. → pages 41

[34] B. Cully, J. Wires, D. T. Meyer, K. Jamieson, K. Fraser, T. Deegan, D. Stodden, G. Lefebvre, D. Ferstay, and A. Warfield. Strata: scalable high-performance storage on virtualized non-volatile memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 17–31. USENIX, 2014. → pages iv, 9, 14, 45, 76

[35] C. Delimitrou, S. Sankar, K. Vaid, and C. Kozyrakis. Decoupling datacenter studies from access to large-scale applications: A modeling approach for storage workloads. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 51–60. IEEE, 2011. → pages 73

[36] P. Denning. working set model of program behavior. *Communications of the ACM*, 1968. → pages 47, 97

[37] C. Ding. Program locality analysis tool, 2014. https://github.com/dcompiler/loca. Visited July 2017. → pages 62

[38] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 245–257. ACM, 2003. → pages 47, 50, 72

[39] G. Dósa. The tight bound of first fit decreasing bin-packing algorithm is FFD(i) <= 11/9OPT(i) + 6/9. In *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2007. → pages 100

[40] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. *ACM Computing Surveys*, 14(2):287–313, 1982. → pages 105

[41] Z. Drudi, N. J. A. Harvey, S. Ingram, A. Warfield, and J. Wires. Approximating hit rate curves using streaming algorithms. In *Proceedings of the 18th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, APPROX '15, pages 225–241, 2015. → pages 9, 55

[42] D. Eklov and E. Hagersten. StatStack: Efficient modeling of LRU caches. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems & Software*, ISPASS '10, pages 55–65. IEEE, 2010. → pages 53, 72

[43] D. Ellard, J. Ledlie, P. Malkani, and M. I. Seltzer. Passive nfs tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03. USENIX, 2003. → pages 68, 99

[44] EMC. DSSD D5, 2016. https://www.emc.com/en-us/storage/flash/dssd/dssd-d5/index.htm. Visited July 2017. → pages 75

[45] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, 1995. → pages 40

[46] B. Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004 (124):5–, Aug. 2004. → pages 42

[47] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm. In *Proceedings of the 2007 Conference on Analysis of Algorithms*, AofA '07, pages 127–146, 2007. → pages 45, 55

[48] E. Freuder. A sufficient condition for backtrack-free search. *Communications of the ACM*, 29(1):24–32, 1982. → pages 85

[49] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43, 2003. → pages 105

[50] G. A. Gibson, K. Amiri, and D. F. Nagle. A case for network-attached secure disks. Technical Report CMU-CS-96-142, Carnegie-Mellon University.Computer science. Pittsburgh (PA US), Pittsburgh, 1996. → pages 17

[51] A. Goel, C. Shahabi, S.-Y. D. Yao, and R. Zimmermann. SCADDAR: An efficient randomized technique to reorganize continuous media blocks. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE '02, pages 473–482. IEEE, 2002. → pages 105

[52] R. L. Graham. Bounds on multiprocessing anomalies and related packing algorithms. In *Proceedings of the May 16-18, 1972, Spring Joint Computer Conference*, AFIPS '72 (Spring), pages 205–217, New York, NY, USA, 1972. ACM. → pages 86

[53] J. Guerra, H. Pucha, J. S. Glider, W. Belluomini, and R. Rangaswami. Cost effective storage using extent based dynamic tiering. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, pages 273–286. USENIX, 2011. → pages 105

[54] R. Haralick and G. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14(3):263–313, 1980. → pages 85

[55] E. Hebrard, B. Hnich, B. O'Sullivan, and T. Walsh. Finding diverse and similar solutions in constraint programming. In *Proceedings of the 20th National Conference on Artificial Intelligence*, AAAI '05, pages 372–377. MIT Press, 2005. → pages 85

[56] D. Hildebrand and P. Honeyman. Exporting storage systems in a scalable manner with pnfs. In *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, MSST '05, 2005. → pages 42

[57] R. J. Honicky and E. L. Miller. A fast algorithm for online placement and reorganization of replicated data. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, page 57. IEEE Computer Society, 2003. → pages 105

[58] R. J. Honicky and E. L. Miller. Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution. In *Proceedings of the 18th International Symposium on Parallel and Distributed Processing*, IPDPS '04. IEEE Computer Society, 2004. → pages 105

[59] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC '10. USENIX Association, 2010. → pages 80

[60] N. C. Hutchinson and L. L. Peterson. The x-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan. 1991. → pages 21

[61] B. Jacob, P. Larson, B. Leitao, and S. da Silva. SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems. *IBM Redbook*, 2008. → pages 46

[62] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 14–24. ACM, 2006. → pages 50

[63] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement*, IMC '09, pages 202–208, New York, NY, USA, 2009. ACM. → pages 105

[64] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM. → pages 24, 105

[65] S. R. Kashyap, S. Khuller, Y.-C. J. Wan, and L. Golubchik. Fast reconfiguration of data placement in parallel disks. In *Proceedings of the Meeting on Algorithm Engineering and Experiments*, ALENEX '06, pages 95–107. SIAM, 2006. → pages 88

[66] S. Khuller, Y. A. Kim, and Y.-C. J. Wan. Algorithms for data migration with cloning. In *Proceedings of the 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '03, pages 27 – 36, 2003. → pages

[67] S. Khuller, Y.-A. Kim, and A. Malekian. Improved approximation algorithms for data migration. *Algorithmica*, 63(1-2):347–362, 2012. → pages 88

[68] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004. → pages 47

[69] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000. → pages 21

[70] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and

B. Zhao. OceanStore: an architecture for global-scale persistent storage. *SIGPLAN Notices*, 35(11):190–201, 2000. → pages 105

[71] E. K. Lee and C. A. Thekkath. Petal: distributed virtual disks. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 84–92, 1996. → pages 41

[72] A. W. Leung, S. Pasupathy, G. R. Goodson, and E. L. Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX Annual Technical Conference*, ATC '08, pages 213–226, 2008. → pages 68

[73] L. Lin, Y. Zhu, J. Yue, Z. Cai, and B. Segee. Hot random off-loading: A hybrid storage system with dynamic data migration. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer Telecommunication Systems*, MASCOTS '11, pages 318–325. IEEE Computer Society, 2011. → pages 105

[74] Linux Device Mapper Resource Page. http://sourceware.org/dm/. Visited July 2017. → pages 21

[75] Linux Logical Volume Manager (LVM2) Resource Page. http://sourceware.org/lvm2/. Visited July 2017. → pages 21

[76] T. Luo, S. Ma, R. Lee, X. Zhang, D. Liu, and L. Zhou. S-cave: Effective ssd caching to improve virtual machine storage performance. In *Parallel Architectures and Compilation Techniques*, PACT '13, pages 103–112, 2013. → pages 41

[77] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. → pages 85

[78] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970. → pages 46, 47, 50, 62, 72

[79] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, 2003. → pages 70

[80] D. T. Meyer, B. Cully, J. Wires, N. C. Hutchinson, and A. Warfield. Block mason. In *Proceedings of the First Conference on I/O Virtualization*, WIOV '08, 2008. → pages 21

[81] D. T. Meyer, B. Cully, J. Wires, N. C. Hutchinson, and A. Warfield. Block mason. In *First Workshop on I/O Virtualization*, WIOV '08. USENIX Association, 2008. → pages 9

[82] D. T. Meyer, M. Shamma, J. Wires, Q. Zhang, N. C. Hutchinson, and A. Warfield. Fast and cautious evolution of cloud storage. In *Proceedings of the 2nd USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '10. USENIX Association, 2010. → pages 9

[83] P. Mochel. The sysfs Filesystem. In *Linux Symposium*, page 313, 2005. → pages 46

[84] D. Mosberger and L. L. Peterson. Making paths explicit in the scout operating system. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 153–167, 1996. → pages 21

[85] M. Nanavati, J. Wires, and A. Warfield. Decibel: Isolation and sharing in disaggregated rack-scale storage. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '17, pages 17–33. USENIX Association, 2017. → pages 9

[86] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):10, 2008. → pages 62

[87] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. I. T. Rowstron. Everest: Scaling down peak loads through i/o off-loading. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 15–28. USENIX Association, 2008. → pages 99, 105

[88] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI '12, pages 1–15, Berkeley, CA, USA, 2012. USENIX Association. → pages 42, 105

[89] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Proceedings of the 2012 IEEE 26th International Parallel & Distributed Processing Symposium*, IPDPS '12, pages 1284–1294. IEEE, 2012. → pages 46, 50, 62, 72

[90] D. Ongaro, S. M. Rumble, R. Stutsman, J. K. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Stystems Principles*, SOSP '11, pages 29–41. ACM, 2011. → pages 105

[91] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman. The case for ramcloud. *Communications of the ACM*, 54(7):121–130, July 2011. → pages 42

[92] C. Petersen. Introducing Lightning: A flexible NVMe JBOF, March 2016. https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/. Visited July 2017. → pages 75

[93] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE Computer Society, 2006. → pages 47

[94] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware 2001, pages 329–350. Springer, 2001. → pages 105

[95] Y. Saito, S. Frølund, A. Veitch, A. Merchant, and S. Spence. Fab: building distributed enterprise disk arrays from commodity components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 48–58, New York, NY, USA, 2004. ACM. → pages 41, 105

[96] J. R. Santos, R. R. Muntz, and B. A. Ribeiro-Neto. Comparing random data allocation and data striping in multimedia servers. In *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '00, pages 44–55. ACM, 2000. → pages 105

[97] T. Schiex, H. Fargier, and G. Verfaillie. Valued constraint satisfaction problems: Hard and easy problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, IJCAI '95, pages 631–639. Morgan Kaufmann, 1995. → pages 85

[98] SCSI Object-Based Storage Device Commands - 2, 2011. http://www.t10.org/members/w_osd-.htm. Visited July 2017. → pages 25

[99] Seagate Kinetic Open Storage Documentation. http://www.seagate.com/ca/en/tech-insights/kinetic-vision-how-seagate-new-developer-tools-meets-the-needs-of-cloud-storage-platforms-master-ti/. Visited July 2017. → pages 17, 25

[100] B. Seo and R. Zimmermann. Efficient disk replacement and data migration algorithms for large disk subsystems. *ACM Transactions on Storage*, 1(3): 316–345, 2005. → pages 105

[101] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating storage for virtual desktops. FAST '11, pages 31–45. USENIX, 2011. → pages 9, 68, 99

[102] X. Shen, Y. Zhong, and C. Ding. Locality phase prediction. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 165–176. ACM, 2004. → pages 50

[103] X. Shen, J. Shaw, B. Meeker, and C. Ding. Locality approximation using time. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 55–61. ACM, 2007. → pages 50

[104] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society. → pages 105

[105] J. Sievert. Iometer: The I/O performance analysis tool for servers, 2004. http://www.iometer.org. Visited July 2017. → pages 71

[106] A. J. Smith. Two methods for the efficient analysis of memory address trace data. *IEEE Transactions on Software Engineering*, 3(1):94–101, 1977. → pages 50, 73

[107] G. Soundararajan, D. Lupei, S. Ghanbari, A. D. Popescu, J. Chen, and C. Amza. Dynamic resource allocation for database servers running on virtual storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*, FAST '09. USENIX, 2009. → pages 50

[108] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, Aug. 2001. → pages 105

[109] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, 1992. → pages 47, 50, 64

[110] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 313–328. USENIX, 2008. → pages 77, 106

[111] V. Tarasov, S. Kumar, J. Ma, D. Hildebrand, A. Povzner, G. Kuenning, and E. Zadok. Extracting flexible, replayable models from large block traces. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, 2012. → pages 73

[112] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, SOSP '97, pages 224–237, 1997. → pages 41

[113] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the 3rd ACM Symposium on Cloud Computing*, SoCC '12, pages 8:1–8:13, New York, NY, USA, 2012. ACM. → pages 41

[114] X. Wang, Y. Li, Y. Luo, X. Hu, J. Brock, C. Ding, and Z. Wang. Optimal footprint symbiosis in shared cache. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '15, pages 412–422. IEEE Computer Society, 2015. → pages 99

[115] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 307–320. USENIX Association, 2006. → pages 42, 105

[116] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM. → pages 105

[117] S. A. Weil, A. W. Leung, S. A. Brandt, and C. 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, pages 35–44. ACM Press, 2007. → pages 42, 105

[118] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: A scalable isolation kernel. In *Proceedings of the 10th ACM SIGOPS European Workshop*, 2002. → pages 40

[119] J. Wilkes, R. A. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. *ACM Transactions on Computer Systems*, 14 (1):108–136, 1996. → pages 105

[120] J. Wires and A. Warfield. Mirador: An active control plane for datacenter storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, FAST '17, pages 213–228. USENIX Association, 2017. → pages iv, 9, 75

[121] J. Wires, M. Spear, and A. Warfield. Exposing file system mappings with mapfs. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage '11. USENIX Association, 2011. → pages 9

[122] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI '14, pages 335–349. USENIX Association, 2014. → pages iv, 9, 44, 81

[123] J. Wires, P. Ganesan, and A. Warfield. Sketches of space: Ownership accounting for shared storage. In *Proceedings of the 8th ACM Symposium on Cloud Computing*, SoCC '17. ACM, 2017. → pages 9

[124] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *USENIX Annual Technical Conference*, ATC '02, pages 161–175. USENIX, 2002. → pages 97

[125] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 350–360. IEEE, 2011. → pages 62, 72

[126] X. Xiang, C. Ding, H. Luo, and B. Bao. HOTL: a higher order theory of locality. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 343–356. ACM, 2013. → pages 47, 73

[127] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 3–10, Berkeley, CA, USA, 2012. USENIX Association. → pages 41

[128] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 103–116. ACM, 2006. → pages 50

[129] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *Annual Technical Conference*, ATC '11, pages 223–228. USENIX, 2011. → pages 50

[130] W. Zheng and G. Zhang. Fastscale: Accelerate RAID scaling by minimizing data migration. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, pages 149–161. USENIX, 2011. → pages 105

[131] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 255–266. ACM, 2004. → pages 50

[132] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 177–188. ACM, 2004. → pages 47, 50

[133] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX Annual Technical Conference*, ATC '01, pages 91–104, 2001. → pages 48