Towards a Storage Stack for the Data Center

by

Ioan Alexandru Stefanovici

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto

Abstract

Towards a Storage Stack for the Data Center

Ioan Alexandru Stefanovici
Doctor of Philosophy
Graduate Department of Computer Science
University of Toronto
2016

The storage stack in a data center consists of all the hardware and software layers involved in processing and persisting data to durable storage. The shift of the world's computation to data centers is placing significant strain on the storage stack, leading to a stack that is unreliable and non-performant. This is caused in large part by a lack of understanding of the failure and performance characteristics of critical hardware components, and a lack of programmability and control over the numerous software layers in the stack. The broad goal of this thesis is to improve the storage stack by leveraging insights gained from empirical studies of real-world production systems, and by proposing a new paradigm for implementing and enhancing distributed storage functionality that enables the vertical specialization of the storage stack to a wide variety of customer and data center provider needs.

The first part of this thesis studies the reliability of main memory in large-scale production systems. Our findings show that conventional wisdom about memory reliability is incorrect, and that physical hardware is in fact the main culprit for most errors in main memory in the field. As a result, existing memory error protection mechanisms are inadequate. We then use the insights gained from the empirical study to propose and evaluate a suitable error protection mechanism for future data centers.

The second part of this thesis offers an empirical study of the effects of temperature on the performance and power consumption of the storage stack. Since cooling constitutes a large fraction of the total cost of ownership in a data center, increasing temperatures in a data center without sacrificing performance can have a huge impact on the power consumption and carbon footprint of data centers.

The final part of this thesis proposes a new paradigm for implementing and enhancing distributed storage functionality by creating programmable APIs that allow dynamic configuration and control of the software stages along the storage stack, and designing and implementing an IO routing primitive for the storage stack.

Acknowledgements

This thesis and who I am wouldn't exist without a great number of people who have been a part of my life, shaped who I am, and what I know. While no list of them will be exhaustive, any omissions are completely unintentional.

I would like to thank Bianca Schroeder and Eno Thereska for advising me through different parts of my PhD. I thank Bianca for taking me on as her PhD student, teaching me the fine points of working with data from real systems in the field, teaching me how to present my research, and for helping me explore the research community to find my place in it. I thank Eno for being a fantastic mentor while I was interning at Microsoft Research Cambridge. For always going out of his way to share his technical and career experience with me, and for being patient and supportive while I was finding my way around systems-building, and I was learning from my mistakes. I hope to collaborate with him again someday.

Thank you to Angela Demke-Brown for always being there and having her door open, and for the many enjoyable and insightful conversations I've had with her over the years, about systems, papers, teaching, and life. They are something I will miss from U of T. I want to thank Ashvin Goel, Michael Stumm, Ding Yuan, and Dushyanth Narayanan for their feedback on this thesis and the work in it, and for all the fun and engaging conversations at conferences, courses, and seminars over the years. I'm also grateful to the rest of the Systems & Networking Group at Microsoft Research Cambridge for providing me with an engaging and stimulating environment during my time there.

Thank you to Karen Reid and the rest of the teaching faculty in the department for their unshakable optimism and positive attitude. And thank you also to the wonderful staff in the department: Joseph Rhagubar, Lynda Barnes, Vinita Khrishnan, Eric Logan, Vlad Sekulic, Marina Haloulos, Sara Burns, Sarah Lavoie, Vivian Hwang, Miriam Irvine, and Margaret Meaney.

Thank you to my past and current labmates: Andy Hwang, George Amvrosiadis, Daniel Fryer, Phillipa Gill, James Gleeson, Nosayba El-Sayed, Bogdan Simion, Amin Tootoonchian, Olya Irzak, Monia Ghobadi, Kangnyeon Kim, Tiangzheng Wang, Suprio Ray, Francis Deslauries, Peter Goodman, and Svitlana Tumanova. No less so, Christina Christodoulakis, Patricia Thaine, Lalla Mouatadid, Marta Garnelo, Aditya Bhargava, Sean Robertson, Arvie Frydenlund, Siavash Kazemain, Jessica David, Joanna Drummond, Mike Glueck, Chris Maddison, David Liu, Orion Buske, Victoria Wilkes, Elizabeth Patitsas, Giovanna Thron, Natan Dubitski. Thank you all for all the laughs, conversations, coffees, teas, lunches, dinners, drinks, parties, and everything else in between. My time in grad school would not have been the same without such good company. Outside of grad school, I owe a lot to Anthony, Amanda, Andre, Komal, Natalie, Jeannine, Dave, Mike, Amber, Zelko, Marina, Gerald, Jing, and Jes. I'm very fortunate to have such good friends in my life, and I look forward to what the future holds.

Thank you to Andrew Petersen for making me aware of my potential in undergrad, and of all the opportunities available to me. I would not be where I am today without his help.

Finally, thank you to my parents, and my grandmother, for always supporting and encouraging me in whatever I chose to do, and for the sacrifices they made to move to Canada, so that I could have better life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Digital storage is a cornerstone of our modern life. We rely on it to store the world's financial data, the sales information of every e-commerce website, personal communication (e-mail, text and video messaging), scientific and medical data, photos, movies, music, as well as all the data backing the multitude of apps we use on a day-to-day basis.

Over the last few years, this data has increasingly resided in public and private data centers. The amount of digital data we are storing is currently doubling every two years [60], and the world's total amount of data is expected to reach 44 trillion gigabytes by the year 2020 [60]. This has been largely driven by the explosive growth of online services and appplications, and the rise of cloud-computing as a cost-effective platform for application development and hosting. Along with the increase in the sheer volume of data, the diversity of the applications operating on it has also increased dramatically. Data centers run a wide variety of applications from social networks, "big data" analytics (e.g.: MapReduce, Spark), enterpise applications, to public and private Platform/Software-as-a-Service platforms. These applications all occupy different points on the spectrum of requirements (durability, consistency, performance, etc.) and types of transformations they perform on their data.

Storage traffic (IO) from applications to durable storage can traverse a large number of hardware and software layers. Hardware stages include main memory, networks, and the durable storage media (e.g.: magnetic hard disk drives, solid state drives). Software stages include caches, file systems, IO schedulers, hypervisors, virus scanning, deduplication, encryption, etc. Collectively, we refer to all the hardware and software layers involved in persisting data to durable storage media as *the storage stack*. The ever-increasing demands for scale and functionality brought on by our increasing reliance on digital storage, and the shift of the bulk of our computation to the data center have put significant strain on the storage stack in recent years.

Given the fundamental limitations on increasing the speed of individual components, the only way to meet the storage and compute demands of modern appplications is to increase the size of the entire system. A key challenge associated with doing so, however, is reliability. As the number of components in systems grows, failures are becoming the norm rather than the exception. Likewise, growing application demands have also necessitated increasing amounts of software in the storage stack, that offer a wide variety of functionality. In many systems, the complexity and size of this software has become unwieldy, leaving these systems hard to program and manage.

The thesis of this work is: **The current data center storage stack is inadequate. The**

biggest barriers to a reliabile and performant storage stack for the data center are: a) a lack of understanding of the failure and performance characteristics of critical hardware components in real-world production systems, and b) a lack of programmability and control over the numerous software layers in the storage stack.

This thesis examines several crucial problems faced by the storage stack in a modern data center. The first half of this thesis focuses on the reliability of the physical memory in the storage stack, and on the inherent costs of maintaining the reliability of data center storage systems. The second half of this thesis focuses on the lack of programmability and manageability of the numerous software layers in the storage stack. It presents a new paradigm for vertically specializing the storage stack by configuring software stages (in particular *caches*) in the stack via programmable APIs, as well as implementing and enhancing distributed storage functionality that relies on dynamic changes to the path of IOs in the system.

Below we provide a brief overview of the rest of the chapters in this thesis.

1.1 Chapter 3: Errors in Main Memory

Main memory is one of the leading hardware causes for machine crashes in today's data centers [124], and one of the most frequently replaced components in large-scale systems [125]. Designing, evaluating and modeling storage systems that are resilient against memory errors requires a good understanding of the underlying characteristics of main memory errors in the field. While there have been a few first studies on DRAM errors in production systems, these have been too limited in either the size of the data set or the granularity of the data to conclusively answer many of the open questions on DRAM errors. Such questions include, for example, the prevalence of soft errors compared to hard errors, or the analysis of typical patterns of hard errors.

In this chapter, we study data on DRAM errors collected on a diverse range of production systems, in total covering nearly 300 Terabyte years of main memory. As a first contribution, we provide a detailed analytical study of DRAM error characteristics, including both hard and soft errors. We find that, contrary to common wisdom a large fraction of DRAM errors in the field can be attributed to hard errors and we provide a detailed analytical study of their characteristics. As a second contribution, the chapter uses the results from the measurement study to identify a number of promising directions for designing more resilient systems and evaluates the potential of different protection mechanism in the light of realistic error patterns. One of our findings is that simple page retirement policies might be able to mask a large number of DRAM errors in production systems, while sacrificing only a negligible fraction of the total DRAM in the system.

1.2 Chapter 4: Impact of Temperature on the Storage Stack

Maintaining the reliability of large-scale storage systems is very costly. One of the biggest contributors to this cost is cooling. Cooling makes up a significant part of the total cost of ownership of a data center. Interestingly, a key aspect of temperature management in a data center has not been well understood: controlling the setpoint temperature at which to run a data center's cooling system. Most data centers set their thermostat based on (conservative) suggestions by manufacturers, as there is limited understanding of how higher temperatures will affect the system. Studies suggest that increasing the temperature setpoint by just one degree could save 2–5% of the energy consumption.

This chapter serves as a guide to data center operators in understanding the tradeoffs between operating at higher temperatures and the inherent performance and power penalties of doing so. Hard disks, memory, and CPUs all employ a number of hardware reliability mechanisms and features intended to maintain the integrity of data at higher temperatures, and protect the hardware against damage or excessive errors. We provide extensive experimental results across a wide array of representative workloads that quantify the range of performance penalties across the entire spectrum of configuration options.

1.3 Chapter 5: Software-Defined Caching

In data centers, caches work both to provide low IO latencies and to reduce the load on the back-end network and storage. But caches today are not designed for multi-tenancy; system level caches today cannot be configured to match tenant or provider objectives. Exacerbating the problem is the increasing number of un-coordinated caches on the IO data plane. An IO request from an application or VM passes through at least three levels of independent caches until it reaches durable storage. The lack of global visibility on the control plane to coordinate this distributed set of caches leads to inefficiencies, increasing cloud provider costs.

This chapter presents Moirai, a tenant- and workload-aware system that allows data center providers to control their distributed caching infrastructure. Moirai can help ease the management of the cache infrastructure and achieve various objectives, such as improving overall resource utilization or providing tenant isolation and QoS guarantees, as we show through several use cases. A key benefit of Moirai is that it is transparent to applications or VMs deployed in data centers. Our prototype runs unmodified OSes and databases providing immediate benefit to existing applications.

1.4 Chapter 6: Treating the Storage Stack Like a Network

IO from an application to distributed storage traverses not only the network, but also several software stages with diverse functionality. In a typical data center, the number of these stages is often larger than the number of network hops to the destination. Yet, while packet routing is fundamental to networks, no notion of IO routing exists on the storage stack. The path of an IO to an endpoint is predetermined and hard-coded. This forces IO with different needs (e.g., requiring different caching or replica selection) to flow through a one-size-fits-all IO stack structure, resulting in an ossified IO stack.

This chapter presents sRoute, an architecture that provides a routing abstraction for the storage stack. sRoute comprises a centralized control plane and "sSwitches" on the data plane. The control plane sets the forwarding rules in each sSwitch to route IO requests at runtime based on application-specific policies. The architecture works with unmodified applications and VMs, and brings significant benefits to data centers. Customized IO routing leads to a factor of ten improvement for IO tail latency, more than 60% better throughput for a customized replication protocol, and a factor of two improvement in throughput for customized caching.

1.5 Bibliographical Notes

The work presented in this thesis has been published in peer-reviewed computer systems conferences. Each chapter of the thesis corresponds roughly to one conference publication:

- Chapter 3 corresponds to *Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design* [59], co-authored with Andy Hwang, and Bianca Schroeder.

- Chapter 4 corresponds to *Temperature Management in Data Centers: Why Some (Might) Like It Hot* [40], co-authored with Nosayba El-Sayed, George Amvrosiadis, Andy Hwang, and Bianca Schroeder.

- Chapter 5 corresponds to *Software-Defined Caching: Managing Caches in Multi-Tenant Data Centers* [137], co-authored with Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Ant Rowstron, and Tom Talpey.

- Chapter 6 corresponds to *sRoute: Treating the Storage Stack Like a Network* [136], co-authored with Bianca Schroeder, Greg O'Shea, and Eno Thereska.

Chapter 2

Related Works

2.1 Reliability of Main Memory

Several studies [124, 125, 99] have shown that main memory is one of the leading causes of hardware problems that result in machine crashes, and one of the most frequently replaced components in modern large-scale computer systems.

Existing work focuses on understanding the characteristics, prevelance, and effects of memory errors on systems, however it focuses primarily on soft errors [85, 164, 105, 1, 143, 144, 37, 104, 72, 163]. Much of the work done by IBM relies on controlled lab experiments [105, 163, 162, 165], and focuses primarily on soft errors. Field studies of real production systems [85, 81] also focus mostly on soft errors, since they are often assumed to be the leading type of memory errors [36].

More recent exceptions include a field study of Google's data centers [126] that suggests hard errors might be the dominant type of memory errors, based on memory error counts. However, without further data on physical memory error location, it cannot substantiate this claim. Similarly, Li et al. [80] analyze field data collected from 200 production machines and find evidence for hard memory errors on 12 machines, making it hard to draw any statistically significant conclusions. Our work builds to fill the gaps left by these studies, generating statistically significant conclusions about the prevalence and characteristics of memory errors, as well as potential solutions for mechanisms to mitigate the effects of these errros.

Previous work has studied mechanisms to mitigate the effects of memory errors on systems. Much of it was predicated on the assumption that soft errors were the dominant type of memory errors in systems. As a result, one of the cheapeast and most common forms of error correction is SEC-DED (Single Error Correct, Double Error Detect) [83] for individual DIMMs, which can correct single bit flips, and detect double bit flips. A more advanced form of ECC patented by IBM is called Chipkill [35], and can tolerate the failure of an entire DRAM chip inside a DIMM, by distributing consecutive data bits to different DRAM chips. Naturally, this does not come for free, as constant data reconstruction comes at the cost of increased power consumption and somewhat degraded performance. Researchers have also shown how to provide "virtualized" ECC at the software level by storing the checksum information in main memory itself [161].

At the OS level, previous work has investigated page retirement [142, 21, 118], which was implemented in Solaris [142, 21], and can optionally be added to Linux [118]. However, despite the presence of these mechanisms, there are no associated policies for when they should be deployed, whose effectiveness has been thoroughly validated using error traces from real production systems. Our work fills this gap.

2.2 Impact of Temperature on The Storage Stack

Previous work has focused on different aspects of data center cooling and temperature management. One approach is through the careful design and modelling of the physical rooms themselves to improve airflow cooling efficiency [154, 109, 108, 110, 139]. Here, different hot-aisle/cold-aisle designs are proposed and evaluated through fluid dynamics modeling.

Other work [14, 112, 117, 128, 82] focuses on workload placement, request distribution, and load balancing among different servers in a cluster. The focus here is on assignment of work to different physical machines (as well as scaling the size of available machines in the cluster by turning machines on/off, or putting them in lower power states), with power and temperature management as a first-order optimization goal. Rather than considering clusters as a whole, some examples [44, 47, 102, 90] of other work focus on power reduction features and optimizations at the level of an individual server.

The write-verify command is described in the SCSI reference manual [61]. Riska and Riedel [121] propose an alternative to Read-after-Write called Idle Read After Write (IRAW), which keeps the contents of the write in the disk buffer cache and performs the write verify when the disk is idle, as opposed to on the critical Write path, which may impact user performance. While this approach sounds attractive and superior to Read-after-Write, it introduces possible contention for caches, and potentially for task scheduling.

Miftakhutdinov et al. [97] model the energy savings gained as a result of dynamic voltage frequency scaling (DVFS) in conjunction with its impact on memory performance to create a better predictor for DVFS's use.

2.3 Software-Defined Caching

There has been much work recently on caches in data centers. Much of it focused on specialized *application* caches, such as Facebook's photo-serving stack [58], Facebook's social graph store [16], *memcached* [43], or explicit cloud caching services [25, 24]. In contrast, our work is on *system* storage caches for hosting cloud providers that run arbitrary workloads.

Work on system caches has focused on efficient use of memory for virtual machines through ballooning and sharing techniques [155, 55, 98], which are implemented in state-of-the-art hypervisors like VMware's and Hyper-V. Our work focuses on other caches in the system, beneath the VM abstraction.

Some prior work has focused on isolating the cache effects of streams with different access patterns (sequential versus looping) within the same workload from each other [89, 71, 26, 49]. However, these policies are not workload or tenant aware and cannot prevent a more aggressive workload from occupying more than its fair share of cache. Moreover, each of these policies might actually work better when applied in the context of Moirai, where a cache policy works on per workload segregated cache, as patterns of different workloads don't get interspersed and hence might be easier to detect. Others propose methods to detect changes in workload patterns and dynamically adjust the caching policy used by the system [50]. Moirai provides a perfect vehicle for implementing such an approach and it would be interesting to extend it to support such functionality. Yet another line of work [56] proposes that applications explicitly manage their cache space and its contents, while our goal was to provide a solution that is transparent to the application.

Several other papers have addressed the problem of inefficiencies in cache hierarchies, e.g., some [23, 79] pass hints from the client to better inform caching decisions at the storage server and others [159] extend the SCSI command set by a demote command to avoid double caching. Our goal was a solution that does not require application or VM support, or changes to existing protocols.

Similar to recent work on software-defined networking (SDNs) [73, 20, 160, 42, 116, 64, 149] and storage (SDS) [147], our architecture is controller-based with a separation between the data and the control plane. Moirai's implementation uses IOFlow's [147] mechanisms for traffic classification, however Moirai's implementation required extensions to IOFlow, e.g., to support arbitrary inspection and manipulation of IO request data, as well as the implementation of the three core components Moirai comprises (as described in Section 5.2 and Section 5.4).

2.4 Treating the Storage Stack Like a Network

Our work is most related to software-defined networks (SDNs) [73, 20, 160, 42, 116, 64, 149, 30] and software-defined storage (SDS) [4, 147]. Specifically, our work builds directly upon the control-data decoupling enabled by IOFlow [147], and borrows two specific primitives: classification and rate limiting based on IO headers for quiescing. IOFlow also made a case for request routing. However, it only explored the concept for bypassing stages along the path, and did not consider the full IO routing spectrum where the path and endpoint can also change, leading to consistency concerns. This chapter provides the full routing abstraction.

There has been much work in providing applications with specialized use of system resources [41, 67, 12, 4, 6]. The Exokernel architecture [41, 67] provides applications direct control over resources with minimal kernel involvement. SPIN [12] and Vino [127] allow applications to download code into the kernel, and specialize resource management for their needs. Another orthogonal approach is to extend existing OS interfaces and pass hints vertically along the IO stack [4, 6, 5, 91]. Hints can be passed in both directions between the application and the system, exposing application needs and system resource capabilities to provide a measure of specialization.

In contrast to the above approaches, this chapter makes the observation that modern IO stacks support mechanisms for injecting stages with specialized functionality (e.g., in Windows [95], FreeBSD [46] and Linux [84]). sRoute transforms the problem of providing application flexibility into an IO routing problem. sRoute provides a control plane to customize an IO stack by forwarding IO to the right stages without changing the application or requiring a different OS structure.

We built three control applications on top of IO routing. The functionality provided from each has been extensively studied in isolation. For example, application-specific file cache management has shown significant performance benefits [19, 56, 74, 159, 58, 137]. Snapshots, copy-on-write and file versioning all have at their core IO routing. Hard-coded implementations can be found in file systems like ZFS [107], WAFL [57] and btrfs [122]. Similarly, Narayanan et al. describe an implementation of load balancing through IO offloading of write requests [101, 100]. Abd-el-malek et al. describe a system implementation where data can be re-encoded and placed on different servers [2]. Finally, several distributed storage systems each offer different consistency guarantees [7, 28, 145, 34, 78, 29, 75, 146, 17, 22]. In contrast to these specialized implementations, sRoute offers a programmable IO routing abstraction that allows for all this functionality to be specified and customized at runtime.

Chapter 3

Errors in Main Memory

3.1 Introduction

Recent studies point to main memory as one of the leading hardware causes for machine crashes and component replacements in today's data centers [142, 124, 99]. As the amount of DRAM in servers keeps growing and chip densities increase, DRAM errors might pose an even larger threat to the reliability of future generations of systems.

As a testament to the importance of the problem, most server systems provide some form of protection against memory errors. Most commonly this is done at the hardware level through the use of DIMMs with error correcting codes (ECC). ECC DIMMs either provide single-bit error correction and double-bit error detection (SEC-DED); or use more complex codes in the chipkill family [35] that allow a system to tolerate an entire chip failure at the cost of somewhat reduced performance and increased energy usage. In addition, some systems employ protection mechanisms at the operating system level. For example, Solaris tries to identify and then retire pages with hard errors [142, 21]. Researchers have also explored other avenues, such as virtualized and flexible ECC at the software level [161]. In contrast to server systems, most consumer-grade systems provide no protection against memory errors, so any error can potentially lead to corrupted data or a machine crash.

The effectiveness of different approaches for protecting against memory errors and the most promising directions for designing future systems that are resilient in the face of increased DRAM error rates depend greatly on the nature of memory errors. For example, SEC-DED based ECC is most effective for protecting against transient random errors, such as soft errors caused by alpha particles or cosmic rays. Conversely, mechanisms such as page retirement are only effective in protecting against hard errors, which are due to physical device defects and tend to be repeatable. In general, any realistic evaluation of system memory reliability relies on accurate information about the underlying error process, including the relative frequency of hard versus soft errors, and the typical modes of hard errors (e.g. device defects affecting individual cells, whole rows, columns, or an entire chip).

While there exists a large body of work on protecting systems against DRAM errors, the nature of DRAM errors is not very well understood. Most existing work focuses solely on soft errors [85, 164, 105, 1, 143, 144, 37, 104, 72, 163], as soft error rates are often assumed to be orders of magnitudes greater than typical hard error rates [36]. However, there are no large-scale field studies backing up this assumption. Existing studies on DRAM errors are quite old and rely on controlled lab experiments, rather than production machines [105, 163, 162, 165], and focus on soft errors only [81]. One exception is a study by Li et al. [80], which analyzes field data collected on 200 production machines and finds evidence that the rate of hard errors might be higher than commonly assumed. However, the limited size of their data set, which includes only a total of 12 machines with errors, makes it hard to draw statistically significant conclusions on the rate of hard versus soft errors, or common modes of hard errors. Another field study [126] speculates that the rate of hard errors might be significant based on correlations they observe in error counts over time, but it lacks finer-grained data such as information on the physical location errors, which would permit the study of error patterns and the likely frequency of hard versus soft errors, in order to validate their hypothesis.

The goal of this chapter is two-fold. First, we strive to fill the gaps in our understanding of DRAM error characteristics, in particular the rate of hard errors, their patterns, and their observed effects in the field. Towards this end, we provide a large-scale field study based on a diverse range of production systems, covering nearly 300 Terabyte-years of main memory. The data includes detailed information on the location of errors, which allows us to statistically conclusively answer several important open questions about DRAM error characteristics. In particular, we find that a large fraction of DRAM errors in the field can be attributed to hard errors and we provide a detailed analytical study of their characteristics.

As a second contribution, we use the results from the measurement study to identify a number of promising directions for designing more resilient systems and evaluate the potential of different protection mechanisms in the light of realistic error patterns. One of our findings is that simple page retirement policies, which are currently not widely used in practice, might be able to mask a large number of DRAM errors in production systems, while sacrificing only a negligible fraction of the total system's DRAM.

3.2 Study Overview

3.2.1 Overview of data and systems

Our study is based on data from four different environments: the IBM Blue Gene/L (BG/L) supercomputer at Lawrence Livermore National Laboratory, the Blue Gene/P (BG/P) from the Argonne National Laboratory, a high-performance computing cluster at the SciNet High Performance Computing Consortium, and 20,000 machines randomly selected from Google's data centers. Below we provide a brief description of each of the systems and the data we obtained.

BG/L: Our first dataset is from the Lawrence Livermore National Laboratory's IBM Blue Gene/L (BG/L) supercomputer. The system consists of 64 racks, each containing 32 node cards. Each node card is made up of 16 compute cards, which are the smallest replaceable hardware component for the system; we refer to the compute cards as "nodes". Each compute card itself contains two PowerPC 440 cores, each with their own associated DRAM chips, which are soldered onto the card; there is no notion of a "DIMM" (see [48] for more details).

We obtained BG/L logs containing data generated by the system's RAS infrastructure, including count and location messages pertaining to correctable memory errors that occur during a job and are reported upon job completion.

The BG/L memory port contains a 128-bit data part that's divided into 32 symbols, where the ECC is able to correct any error pattern within a single symbol, assuming no errors occur in any other symbols. However, the system can still function in the event of two symbols with errors by remapping one of the symbols to a spare symbol, and correcting the other with ECC [106].

Due to limitations in the types of messages that the BG/L log contains, we are only able to report on multi-bit errors that were detected (and corrected) within a single symbol. As such, we refer to these as MBEs (multi-bit errors) for the BG/L system throughout the chapter. However it's worth noting that 350 compute cards (20% of all compute cards with errors in the system) reported activating symbol steering to the spare symbol. This is indicative of more severe errors that required more advanced ECC technologies (like bit-sparing) to correct. In addition, a cap was imposed on the total count of correctable errors accumulated during a job for part of the dataset, making our results for both multi-bit errors and total correctable error counts very conservative compared to the actual state of the system.

BG/P: The second system we studied is the Blue Gene/P (BG/P) from the Argonne National Laboratory. The system has 40 racks containing a total of 40,960 compute cards (nodes). Each node in BG/P has four PowerPC 450 cores and 40 DRAM chips totalling 2GB of addressable memory. As the successor to BG/L, BG/P has stronger ECC capabilities and can correct single and double-symbol errors. The system is also capable of chipkill error correction, which tolerates failure of one whole DRAM chip [66].

We obtained RAS logs from BG/P reporting correctable error samples. Only the first error sample on an address during the execution of a job is reported, and total occurrences for each error type summarized at the end. Due to the sampling and counter size, the number of correctable errors is once again very conservative. However, the correctable samples provide location information which allows us to study the patterns and physical distribution of errors.

Unlike BG/L, there is no bit position information for single-symbol errors. There is no way to determine the number of bits that failed within one symbol. Therefore, we report single-symbol errors as single-bit errors and double-symbol errors as multi-bit errors, and refer to the latter as MBEs for the BG/P system. A double-symbol error is guaranteed to have at least two error bits that originate from the pair of error symbols. This is once again an under-estimation of the total number of multi-bit errors.

SciNet: Our third data source comes from the General Purpose Cluster (GPC) at the SciNet High Performance Computing Consortium. The GPC at SciNet is currently the largest supercomputer in Canada. It consists of 3,863 IBM iDataPlex nodes, each with 8 Intel Xeon E5540 cores and 16GB of addressable memory that uses basic SEC-DED ECC. The logs we collected consist of hourly-dumps of the entire PCI configuration space, which expose the onboard memory controller registers containing counts (with no physical location information) of memory error events in the system.

Google: Our fourth data source comes from Google's datacenters and consists of a random sample of 20,000 machines that have experienced memory errors. Each machine comprises a motherboard with some processors and memory DIMMs. The machines in our sample come from 5 different hardware platforms, where a platform is defined by the motherboard and memory generation. The memory in these systems covers a wide variety of the most commonly used types of DRAM. The DIMMs come from multiple manufacturers and models, with three different capacities (1GB, 2GB, 4GB), and cover the three most common DRAM technologies: Double Data Rate (DDR1), Double Data Rate 2 (DDR2) and Fully-Buffered (FBDIMM). We rely on error reports provided by the chipset. Those reports include accurate accounts of the total number of errors that occurred, but due to the limited number of registers available for storing addresses affected by errors only provides samples for the addresses of errors. For this reason, the number of repeat errors we observe and the probability of errors repeating are very conservative estimates, since there might be repeat errors that we missed because they were not sampled.

3.2.2 Methodology

A memory error only manifests itself upon an access to the affected location. As such, some systems employ a *memory scrubber* (a background process that periodically scans through all of memory) to proactively detect errors before they are encountered by an application. However, except for some of the Google systems, all the systems we study rely solely on application-level memory accesses without the use of a scrubber.

Categorizing errors observed in the field as either hard or soft is difficult as it requires knowing their root cause. Obtaining a definite answer to the question of whether an observed error is hard and what type of hard error it is (e.g. a stuck bit or a bad column) would require some offline testing of the device in a lab or at least performing some active probing on the system, e.g. by running a memory test after each error occurrence to determine whether the error is repeatable. Instead we have to rely on observational data, which means we will have to make some assumptions in order to classify errors. Matters are complicated further by the fact that many hard errors start out as intermittent errors and only develop into permanent errors over time.

The key assumption that we rely on in our study is that repeat errors at the same location are likely due to hard errors since it would be statistically extremely unlikely that the same location would be hit twice within our measurement period by cosmic rays or other external sources of noise. We therefore view such repeat errors as likely being caused by hard errors. Note however that in practice, hard errors manifest themselves as intermittent rather than on every access to a particular memory location.

We consider different granularities for locations at which errors can repeat. We start by looking at repeats across nodes, but then mainly focus on locations identified by lower-levels in the hardware. To explain our analysis methodology and our findings, we first provide some background on memory hardware. A DIMM comprises multiple DRAM chips, and each DRAM chip is organized into multiple banks, typically 8 in today's systems. A bank consists of a number of two-dimensional arrays of DRAM cells. A DRAM cell is the most basic unit of storage, essentially a simple capacitor representing one bit. The two dimensions of an array are also referred to as rows and columns. We look at repeats of errors at the level of physical addresses, but also with respect to bank, rows and columns at the chip level.

System	Time (days)	Nodes	# DIMMs	DRAM in system (TB)	TByte years
BG/L	214	32,768	N/A	49	28
BG/P	583	40,960	N/A	80	127
SciNet	211	3,863	31,000	62	35
Google	155	20,000	$\sim$ 130,000	220	93

Table 3.1: Summary of system configurations

System	Nodes with errors	Nodes with chipkill errs	Total # of errors	Failures in time per billion hours of operation
BG/L	1,742 (5.32%)	N/A	$227 \cdot 10^6$	97,614
BG/P	1,455 (3.55%)	1.34%	$1.96 \cdot 10^9$	167,066
SciNet	97 (2.51%)	N/A	$49.3 \cdot 10^6$	18,825
Google	20,000	N/A	$27.27 \cdot 10^9$	N/A

Table 3.2: Summary of high-level error statistics recorded in different systems

3.3 Study of error characteristics

3.3.1 High-level characteristics

We begin with a summary of the high-level characteristics of memory errors at the node level. Table 3.1 presents a summary of system configurations, while Table 3.2 summarizes the prevalence of memory errors in the four different systems. We observe that memory errors happen at a significant rate in all four systems with 2.5-5.5% of nodes affected per system. For each system, our data covers at least tens of millions of errors over a combined period of nearly 300 Terabyte years. In addition to correctable errors (CEs), we also observe a non-negligible rate of "non-trivial" errors, which required more than simple SEC-DED strategies for correction: 1.34% of the nodes in the BG/P system saw at least one error that required chipkill to correct it.

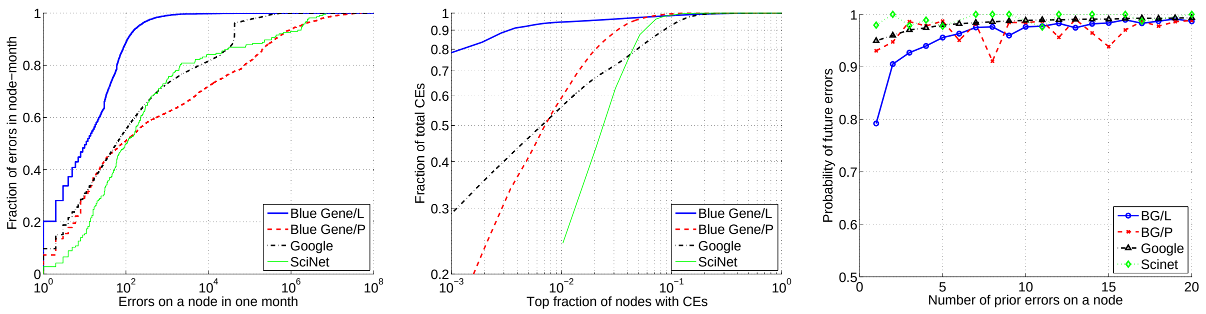


Figure 3.1: The left graph shows the CDF for the number of errors per month per machine. The middle graph shows the fraction y of all errors that is concentrated in the top x fraction of nodes with the most errors. The right graph shows the probability of a node developing future errors as a function of the number of prior errors.

Figure 3.1 (left) and Figure 3.1 (middle) provide a more detailed view of how errors affect the nodes in a system. Figure 3.1 (left) shows the cumulative distribution function (CDF) of the number of errors per node for those nodes that experience at least one error. We see that only a minority (2-20%) of those nodes experience just a single error occurrence. The majority experiences a larger number of errors, with half of the nodes seeing more than 100 errors and the top 5% of nodes each seeing more than a million errors. Figure 3.1 (middle) illustrates how errors are distributed across the nodes within each system. The graph shows for each system the fraction of all errors in the system (X-axis) that is concentrated on just the y% of nodes in the system with the largest number of errors (Y-axis). In all cases we see a very skewed distribution with the top 5% of error nodes accounting for more than 95% of all errors.

Figure 3.1 (left) and (middle) indicate that errors happen in a correlated fashion, rather than independently. This observation is validated in Figure 3.1 (right), which shows the probability of a node experiencing future errors as a function of the number of past errors. We see that even a single error on a node raises the probability of future errors to more than 80%, and after seeing just a handful of errors this probability increases to more than 95%.

The correlations we observe above provide strong evidence for hardware errors as the dominant error mechanism, since one would not expect soft errors to be correlated in space or time. Our observations agree with similar findings reported in [126, 80]. However, the results in [80] were based on a small number of machines (12 machines with errors) and the analysis in [126] was limited to a relatively homogeneous set of systems; all machines in the study were located in Google's datacenters. Our results show that these trends generalize to other systems as well and add statistical significance.

In addition to error counts, the BG systems also record information on the mechanisms that were used to correct errors, which we can use as additional clues regarding the nature of errors. In particular, both BG/P and BG/L provide separate log messages that allow us to distinguish multi-bit errors, and BG/P also records information on chipkill errors (i.e., errors that required chipkill to correct them). We observe that a significant fraction of BG/P and BG/L nodes experience multi-bit errors (22.08% and 2.07%, respectively) and that these errors account for 12.96% and 2.37% of all observed errors, respectively. The fraction of nodes with chipkill errors (only recorded on BG/P) is smaller but still significant, with 1.34% of BG/P nodes affected. Interestingly, while only seen on a small number of nodes, chipkill errors make up a large fraction of all observed errors: 17% of all errors observed on BG/P were not correctable with simple SEC-DED, and required the use of chipkill ECC to be corrected.

These error characteristics motivate us to take a closer look at hard errors and their patterns in the remainder of this chapter.

3.3.2 Error patterns

In this section, we attempt to categorize all banks with errors in our datasets into known error patterns related to hardware defects: single (transient) events, bad cells, bad rows, bad columns, and a whole chip error. A definite answer to the question which category a device with an error falls into would require offline testing of the device in a lab setting. Instead we have to rely on observational data, which means we will have to make a few assumptions when classifying devices. We group all banks that have at least one error into one of the following categories:

repeat address: The bank has at least one error that repeats; i.e., there is at least one address on this bank that is reported twice.

repeat row: The bank has at least one row that has experienced errors at two different locations; i.e., two different addresses on the row.

repeat column: The bank has at least one column that has experienced errors at two different locations; i.e., two different addresses on the column.

corrupt row: The bank has at least one row that has experienced errors at two different addresses on the row and one of these is a repeat address.

corrupt column: The bank has at least one column that has experienced errors at two different addresses on the column and one of these is a repeat address.

single event: These are banks that have only single events; i.e., they have no repeat errors on any of their addresses, rows or columns.

whole chip: These are banks that have a large number of errors (> 100 unique locations) distributed over more than 50 different rows and columns.

Table 3.3 groups each error bank in our dataset into one of the above categories and reports for each system the fraction of banks that falls into each of these categories. Note, that the repeat-address category overlaps with the corrupt row and corrupt column categories. We therefore created an additional entry that reports banks with repeat addresses that do not exhibit corrupt rows or columns.

Error mode	BG/L Banks	BG/P Banks	Google Banks
repeat address	80.9%	59.4%	58.7%
repeat address w/o row/cols	72.2%	30.0%	46.1%
repeat row	4.7%	31.8%	7.4%
repeat column	8.8%	22.7%	14.5%
corrupt row	3.0%	21.6%	4.2%
corrupt column	5.9%	14.4%	8.3%
whole chip	0.53%	3.20%	2.02%
single event	17.6%	29.2%	34.9%

Table 3.3: Frequency of different error patterns

We make a number of interesting observations. The vast majority (65-82%, depending on the system) of all banks experience some form of error pattern that points towards hard errors, i.e. an error pattern other than single events. This observation agrees with the findings in [80], however the conclusions on the frequency of different patterns in [80] are limited due to their small dataset (12 machines with errors). We find that among all error patterns, the single most common one is repeat addresses. Consistently for all systems, more than 50% of all banks with errors are classified as repeat addresses. For all systems, corrupt rows and corrupt columns happen at a significant rate. We note that each system has a clear tendency to develop one type over the other, where the more common type is approximately twice as often observed as the other one. For example, in the case of BG/L and Google, corrupt columns are twice as likely as corrupt rows, while for BG/P it is the other way around. This is likely due to the fact that there are twice as many rows in BG/P banks compared to BG/L banks.

Note that the above numbers on hard error patterns are conservative, and in practice likely higher. Since our observation period for each of the systems is limited and we depend on accesses to memory cells to detect errors, many of the non-repeat errors in our study might eventually have turned out to be repeat errors, but the repeat did not fall within our measurement period. We observe for example that for the systems with shorter observation time (BG/L and Google), the fraction of banks with only repeat addresses, but no bad rows/columns, is higher than in the BG/P system whose data spans a longer observation period. Most likely, the longer observation time increased the chances that a repeat error will manifest and move a repeat row/column to the corrupt row/column category. That indicates that a large fraction of errors we categorize conservatively as repeat rows/columns might actually be true broken rows/columns.

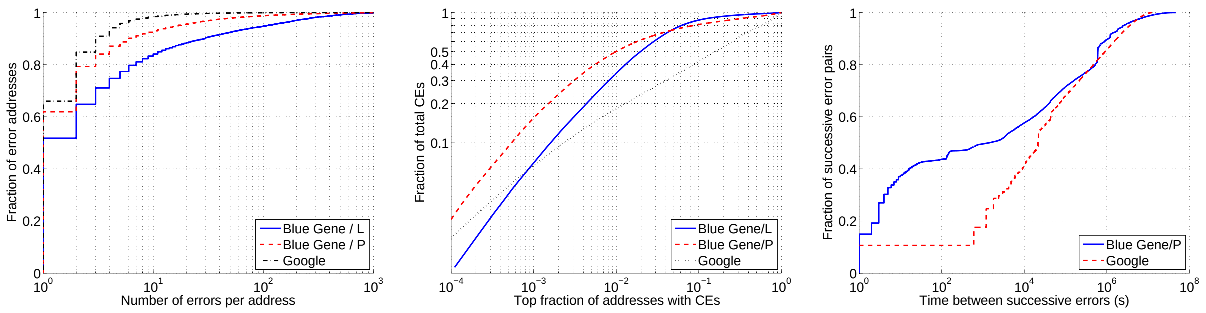


Figure 3.2: The left graph shows the CDF for the number of repeat errors per address (for those addresses with at least one repeat). The middle graph shows the fraction y of all errors that is concentrated in the top x fraction of addresses with the most errors. The right graph shows the CDF of the time between successive errors on an address.

3.3.3 Repeat errors on addresses

The previous section identified repeat addresses as the dominant error pattern, but did not provide any details on their characteristics. The only prior work that hints at repeat addresses as a common error pattern in the field is based on a data set (a dozen machines with errors) that is too small for a detailed study of repeat error characteristics [80]. We therefore study this question in more detail in this subsection.

	BG/L	BG/P	Google
# of error samples	201,206	308,170	1,091,777
# of unique addresses	9,076	44,624	556,161
% unique	4.5	14.5	50.9
% of addresses with repeats	48.2	30.6	32.6
Avg. # of errors / address	44.9	20.3	4.0

Table 3.4: Statistics on repeat addresses

We begin by providing some statistics on the frequency of repeat errors on addresses in Table 3.4. We observe that a high fraction of addresses with errors experience later repeat errors on the same address: a third (for Google) to nearly a half (for BG/L). The average number of repeat errors per address ranges from 4 for Google to as many as 44 for BG/L. For a more detailed view, Figure 3.2 (left) shows the cumulative distribution function (CDF) for the number of repeats per address. Most addresses with repeats (50-60%) see only a single repeat and another 20% see only two repeats. However, the distribution is very skewed with a long tail, where a small fraction of addresses at the end of the tail experiences a huge number of repeats. Figure 3.2 (middle) illustrates the skew in the distribution by plotting the fraction of errors that is made up by the top x% of addresses with the highest error count. It shows that 10% of all addresses with errors account for more than 90% of all observed errors.

When trying to protect against repeat errors it is useful to understand the temporal characteristics of errors. For example, a system using page retirement for pages with hard errors might want to wait before retiring a page that experiences an error until a repeat error occurs, providing some confidence that the problem is indeed due to a hard error. An interesting question is therefore how long it will take before a repeat error happens and an error can confidently be classified as hard. To help answer this question, Figure 3.2 (right) plots the CDF of the time between repeat errors on the same address. The graph shows that, if there is a repeat error it typically happens shortly after the first error occurrence. In BG/P more than half of repeats happen within less than a couple of minutes after the first occurrence. The timing information in the Google data is at a much coarser granularity (recall Section 3.2) and due to sampling we might not see all repeats, which leads to generally longer times until a repeat error shows up. However, we can conclude that more than half of the repeats happen within less than 6 hours. Interestingly, for larger timescales, e.g. on the order of days, where the timing granularity of the Google data should have less of an effect the trends for both systems start to look very similar. In both systems, 90% of all repeat errors are detected within less than 2 weeks.

When interpreting data regarding repeat of errors, it is important to recall that repeat errors (or any errors) are not detected until either the application or a hardware scrubber accesses the affected cell. For the Blue Gene systems, we know that hardware scrubbers are implemented as a feature, but we were not able to determine whether this feature was actually enabled in our systems under study. On the other hand, for the Google machines we know that a subset of them does employ a hardware scrubber that periodically in the background reads through main memory to check for errors. This scrubber reads memory at a rate of 1GB per hour, which means that each memory cell should be touched at least once every day.

To determine how much sooner repeat errors could be detected if a memory scrubber were used we compared the time until a repeat error is detected for those systems with and without the hardware scrubber separately. Interestingly, we find that the use of a scrubber does not significantly reduce the time until a repeat error is detected. Even in the tail of the distribution, where it takes a relatively long time (e.g. several days or more) to identify a repeat error, there is not much difference between systems with and without a scrubber. One possible explanation is that repeat errors might not always be due to stuck bits, where a cell is permanently stuck at a particular value. Instead, they might be due to weaknesses in the hardware that get exposed only under certain access patterns.

3.3.4 Repeat errors within a row/column

Memory errors that are due to hardware problems don't only manifest themselves in the form of repeat errors on the same cell. Section 3.3.2 pointed to repeating errors on different locations within the same row or the same column as another common error mode (recall Table 3.3). This subsection takes a closer look at the characteristics of repeat rows and repeat columns.

We begin by looking at the probability that an error at an individual address will develop into a row or column error. Understanding those probabilities might help predict impending row/column errors, allowing a system to take proactive measures. The three groups of bars in Figure 3.3 (left) summarize our results for columns and Figure 3.3 (right) shows the corresponding results for rows. The gray center bar in each group of bars shows the probability that after an error occurs on an address another error will later develop at a different address along the same column (Figure 3.3 left) and row (Figure 3.3 right), respectively (turning the error from a single event error into a repeat column/row). We observe that in all three cases these probabilities are significant for all systems, with probabilities in the 15% to 30% range.

To put the above repeat probabilities in perspective, we compare them with the unconditional probability of a random bank/row/column developing an error, shown in the black (left-most) bar (the bar is barely visible due to its small magnitude). We see that the unconditional probabilities are orders of magnitude smaller.

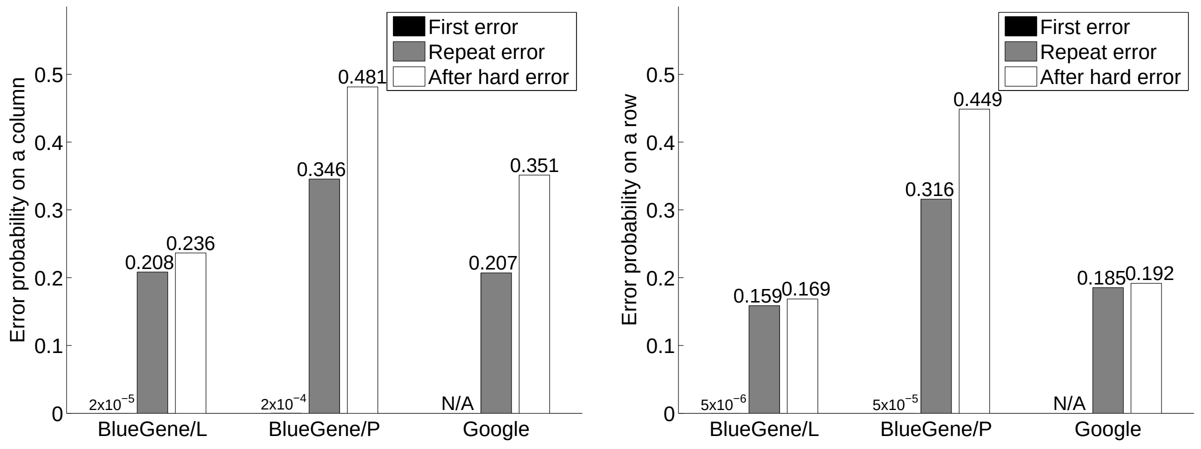


Figure 3.3: The two graphs show the probability that a column (left graph) and a row (right graph) will have an additional error after having one error (middle bar of each group of bars) and the probability that it will have an additional unique error (right bar of each group of bars), i.e. an additional error at an address different from the first.

We also study the probability that a repeat error on an address (rather than a single event error) will turn into a row/column error, i.e. the probability that after a repeat error on an address another address along the same row/column experiences an error. Those probabilities are shown in the white right-most bar in each group of bars in Figure 3.3. In all systems, the presence of a repeat address on a row/column further increases the probability of future errors on this bar/column. In some cases this increase is quite large. For example, for BlueGene/P after observing a repeat error on an address the probability of observing an additional error on another location on the same row/column increases to more than 40%. Again, recall that BG/P is the system with the longest observation period among our datasets, so the probabilities of repeat errors developing into row/column errors effect might be equally strong in the other systems and we might just not observe it due to the shorter timespan of the data.

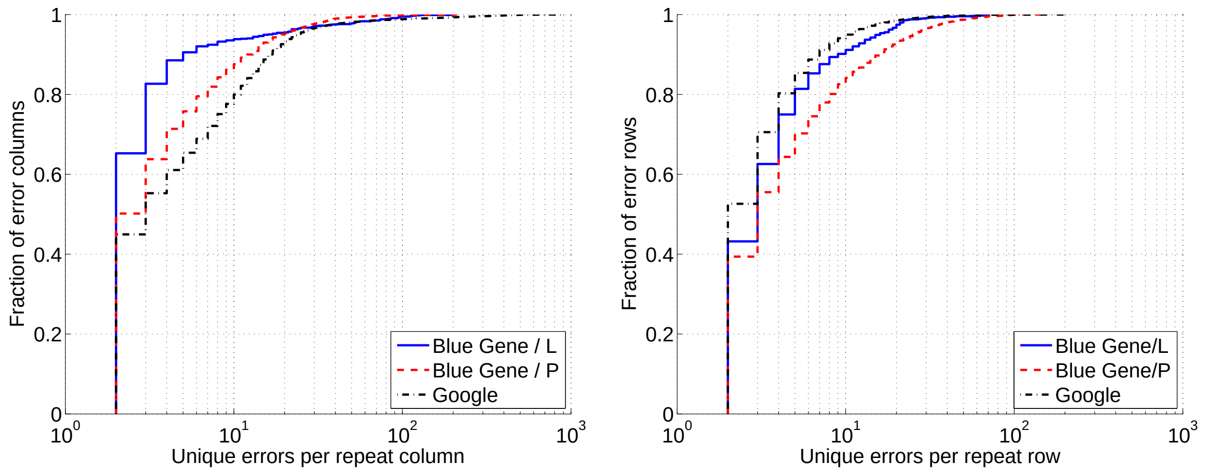


Figure 3.4: Number of errors per repeat row/column

While we have provided information on the probabilities of rows/columns developing multiple errors, another question is how many errors repeat rows and columns typically experience. Figure 3.4 (left) and (right) provide the CDF for the number of unique locations per row/column that experience errors. We see that most rows/columns with multiple errors (40-60%) don't develop more errors beyond the initial 2 errors. However, the top 10-20% of rows/columns develop errors on dozens of unique locations.

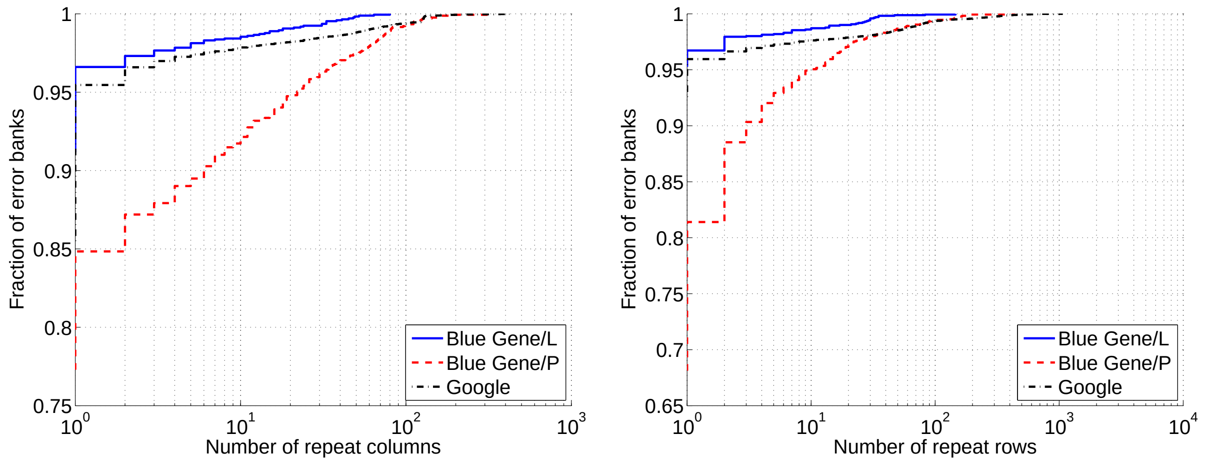


Figure 3.5: Number of repeat rows/columns per bank

Section 3.3.2 showed that a significant number of error banks exhibit row/column errors, but does not quantify the number of repeat rows/columns. We find that the most common case are banks with only a single repeat row/column. Depending on the system, 80-95% of all banks with repeat rows/columns have only a single one (see Figure 3.5). Around 3-8% of banks develop more than 10 repeat columns and 2-5% develop more than 10 repeat rows.

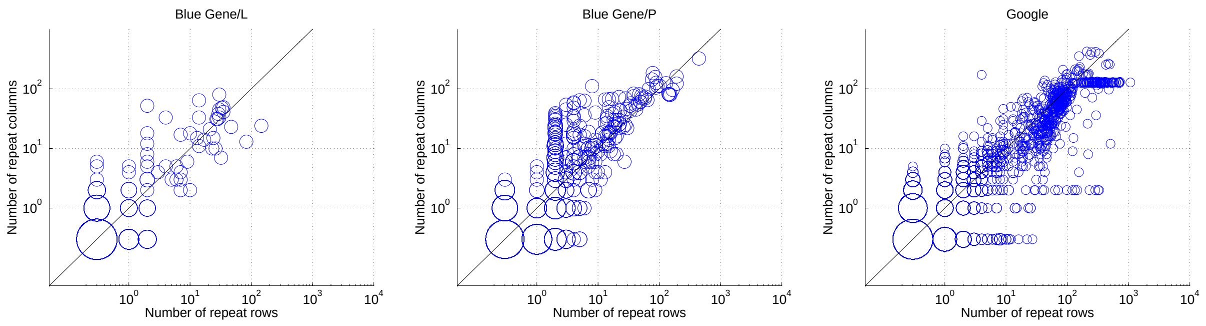


Figure 3.6: The distribution of the number of repeat rows and repeat columns per bank

Interestingly, we observe that a significant number of all banks (3.4%, 17.4%, and 4.4%, for BG/L, BG/P and Google, respectively) experience both repeat rows and repeat columns. To better understand the relationship between the number of repeat rows and columns on a bank, the scatter plot in Figure 3.6 shows for banks with at least one repeat row or column the number of repeat rows and columns. The marker at coordinates (x,y) reflects the fraction of banks that have x repeat columns and y repeat rows. The size of the marker in the scatter plot indicates the fraction of error banks that fall into each of the categories and is chosen to be proportional to the logarithm of the probabilities. For all three systems we observe that banks that have a larger number of repeat rows tend to also have a larger number of repeat columns, and vice versa.

3.3.5 Correlations across rows/columns

While the previous subsection demonstrated that an error on a row or column increases the probability of follow-up errors on the same row/column, it does not tell us anything about correlations between nearby rows/columns, e.g. do multiple errors on a row make it more likely that also some nearby rows will have errors. In this subsection we answer the more general question of how the error probabilities between cells are correlated depending on the physical distance in row and column space between those cells.

The heatmap in Figure 3.7 is an attempt to visualize those correlations. The pixel at the center of the plot at coordinates (0,0) represents a cell with an error. The pixel at coordinates (x,y) represents the probability that the cell that is x columns and y rows away from the original error cell (i.e. at row/column coordinates (a+x, b+y) where (a,b) are the row/column coordinates of the original error) has an error as well. Lighter colors represent higher probabilities.

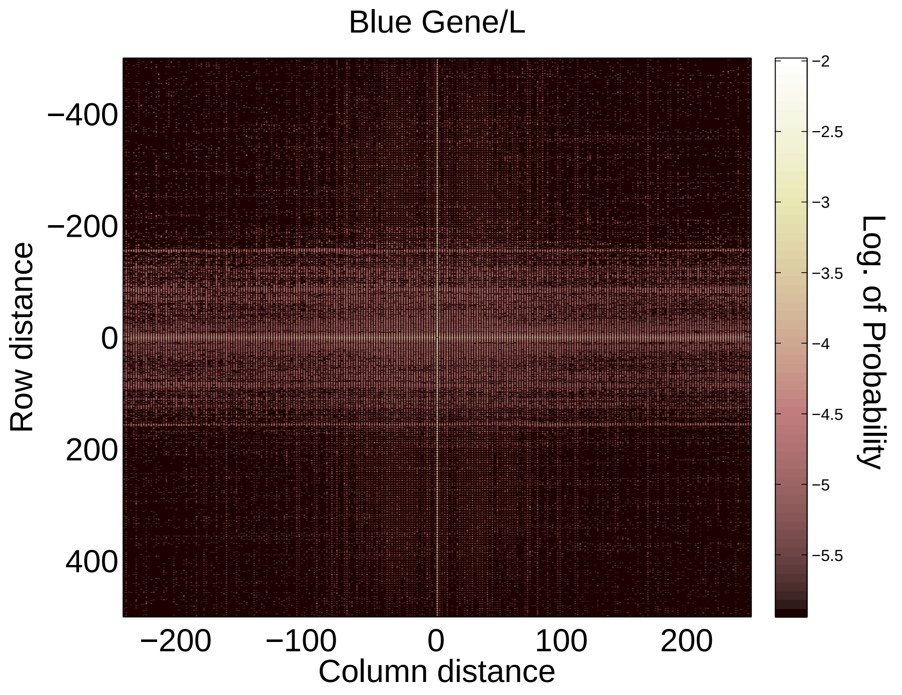
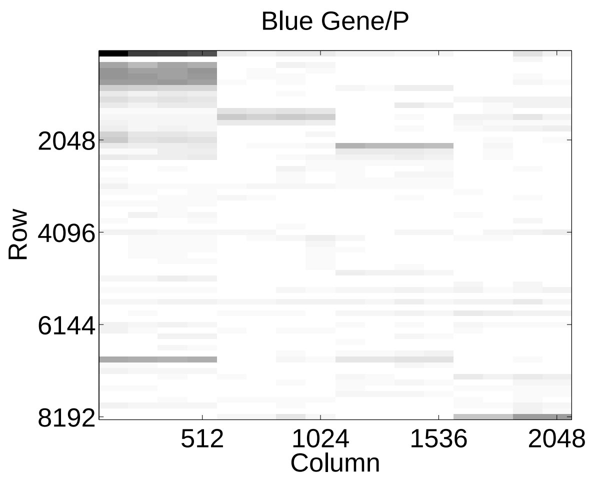


Figure 3.7: A visualization of correlations between errors as a function of their distances in row/column space

Not surprisingly, we observe that cells along the same row or column as the original error cell have increased error probabilities, as indicated by the bright vertical and horizontal line crossing through (0,0). This agrees with our previous observations that errors have a high probability of turning into repeat rows/columns. But we also observe that the error probabilities are increased within a wide band of neighboring rows and columns. For example, a closer study of the error probabilities as a function of row/column distance shows that rows and columns that are within a distance of 10 of the original error have an error probability of 2-5%. While this probability is clearly smaller than that of developing errors on the same row/column, it is significantly larger than that of an average row/column. We also find that the error probabilities show a roughly exponential drop-off as a function of the row/column distance, and that the probabilities are still significantly increased within a range of up to 50-100 rows/columns, compared to an average row/column.

Our study of error probabilities as a function of distance from another error also shows evidence for other patterns, beyond just proximity. In particular, we observe for some systems that cells whose column or row distance from the original error is a multiple of certain powers of two have increased likelihood of errors. Evidence for these regular patterns show up in the heatmap in the form of a grid-like background pattern. By studying the CDF of the pairwise distances between errors, we find for example that for all systems (BG/P, BG/L, Google), cells with distances in row space that are multiples of 4 have noticeably increased error probabilities. Some systems also exhibit other patterns. For example, BG/P also shows clearly increased probabilities at row distances that are multiples of 128.

3.3.6 Error density in different areas of a chip



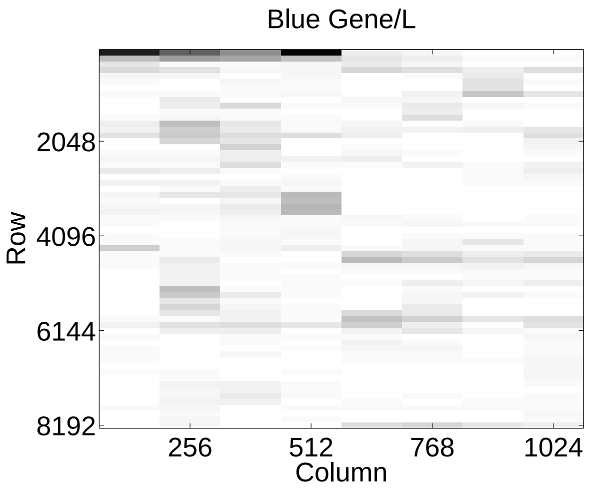


Figure 3.8: The error probabilities for different areas in the row/column space of a bank for BG/L and BG/P.

In this subsection we look at the correlation between errors and their physical location on a chip, i.e. we are asking the question of whether some areas of a chip are more likely than others to experience errors. As before, we use bank, row and column information to distinguish different locations on a chip. We first divide the row/column space of each bank into equal-sized square areas of 128x128 rows/columns, i.e. chunks of size 16KB. We then determine for each of these square areas the probability of observing an error in this area, i.e. the fraction of all unique banks in the system (across all nodes) that have at least one error in this area. Figures 3.8,3.9 show a graphical representation of the results. For this analysis, we report results separately for BG/P and BG/L and for the four different hardware platforms that the Google data covers. Each graph represents the row/column space for one of the systems, each divided into the 128x128 sized squares as described above. Each square is colored according to the observed probability of errors in this area, where darker colors correspond to higher probabilities.

Figures 3.8,3.9 show several interesting trends that are consistent across systems. Focusing on the dark areas in each graph, which present concentrations of errors, we first see that for all systems consistently the top left area shows increased error density. This area tends to span at least the first 512 columns and the first 512 rows. For several of the systems, a whole band along the very top of the row space, across all columns, shows increased error rates. For two of the six systems we observe similar

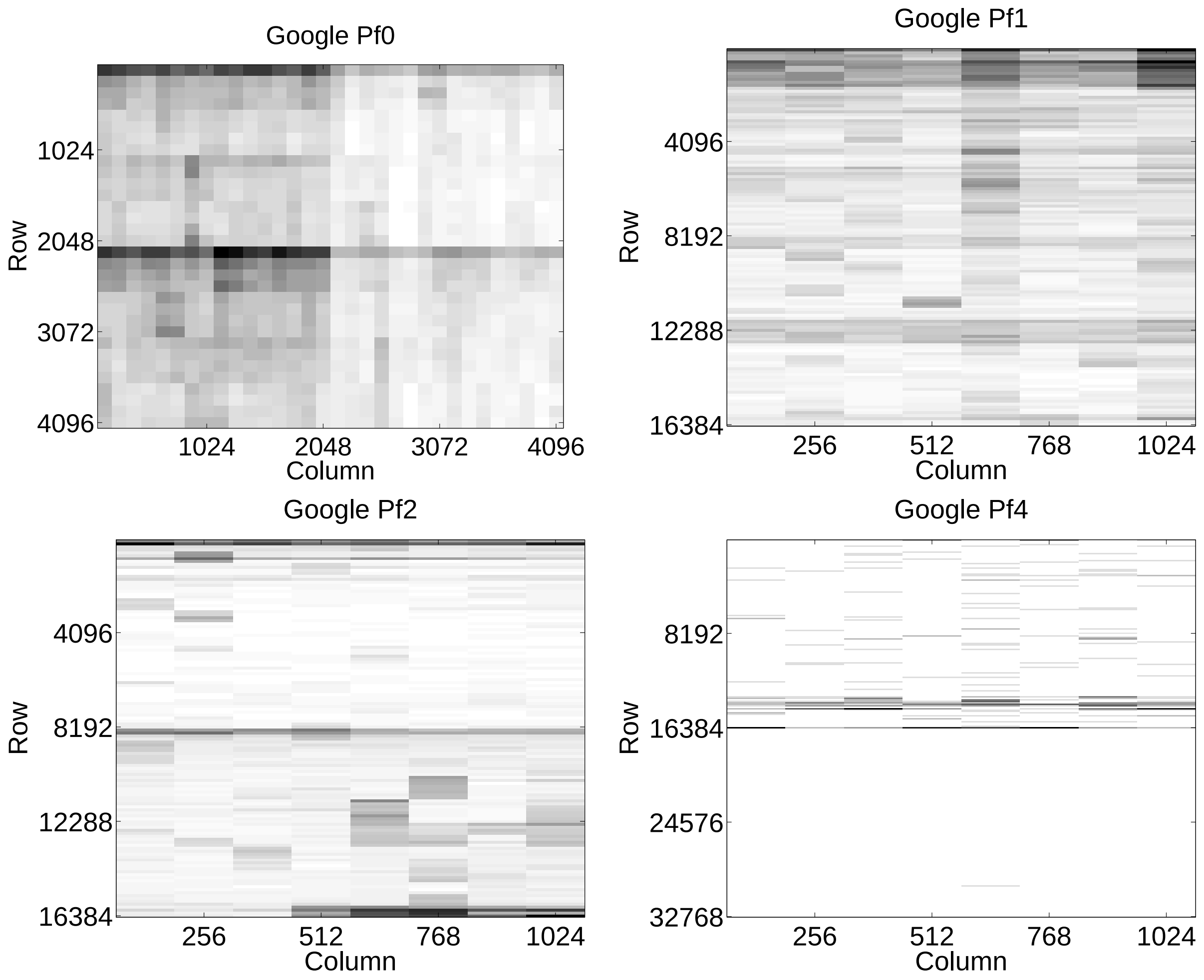


Figure 3.9: The error probabilities for different areas in the row/column space of a bank for Google.

concentrations of errors at the end of the row/column space, i.e. in the bottom right of the graphs at the highest numbered rows and columns. Secondly, we find that for three of the systems the entire rows in the center of the row space exhibit increased error probabilities.

3.3.7 Hard errors from the OS's point of view

Throughout this section we have observed different ways in which DRAM errors tend to cluster in space. We have seen that errors tend to repeat on the same address, along the addresses of a row/column and on certain areas of a chip. All these measures for spatial clustering were very hardware oriented. In order to explore protection mechanisms at the OS level, an important question is how the clustering of errors translates to the operating system level. For example, retiring pages with errors would work most efficiently and effectively if most of the errors tended to cluster on a small number of pages. Unfortunately, error clusters at the hardware level do not directly translate to clusters on pages. For example, errors along the same row or column do not necessarily lie on the same page.

To shed some light on how errors are distributed across pages, Figure 3.10 (left) shows the CDF for the number of errors per page and the number of unique locations with errors per page for those systems

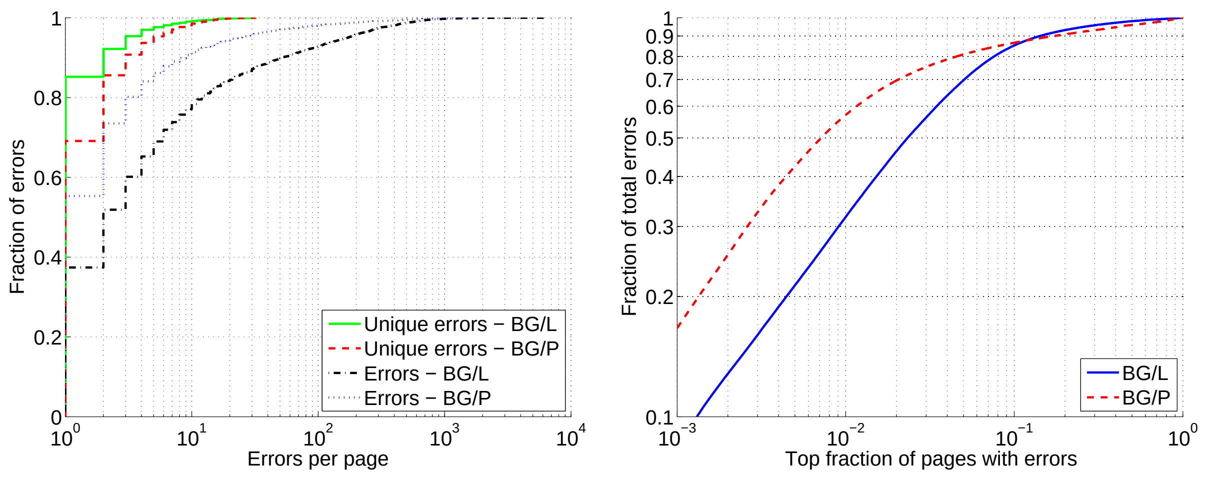


Figure 3.10: Error distribution over pages.

for which the information is available (BG/L and BG/P). The number of unique locations with errors per page is low (on average 1.4 and 1.8 for BG/L and BG/P, respectively) and around 90% of all pages have only a single one. However the total number of errors observed per page is still quite large, most likely due to repeat addresses. More than 60% of the pages experience more than one error, and the average number of errors per page is 31 and 12, for BG/L and BG/P respectively. More importantly, the distribution of errors across pages is very skewed, maybe not surprisingly given the frequency of repeat addresses that we observed earlier. Figure 3.10 (right) shows the fraction of all errors that is contributed by the fraction of the top x% of pages with the most errors. 1% of all pages with errors account for 30-60% of all errors, depending on the system, and the top 10% of all pages with errors account for more than 90% of all errors for both BG/L and BG/P. This skew in the number of errors per page is good news for techniques relying on page retirement, as it means that by retiring a small fraction of pages a large number of errors can be prevented. This observation motivates us to study the possible effectiveness of different page retirement policies in Section 3.4.

3.3.8 Hard errors and multi-bit / chipkill errors

From a systems point of view, the most worrisome types of errors are multi-bit errors, and chipkill errors, as these are the errors that in the absence of sufficiently powerful hardware ECC turn into uncorrectable errors leading to a machine crash (or if undetected, to the use of corrupted data). Given the correlations we observed between errors in the previous subsections, an interesting question is whether it is possible to predict an increased likelihood of future multi-bit or chipkill errors based on the previous error behavior in the system. In particular, one might speculate that prior repeat errors, which likely indicate hard errors, will increase the probability of later multi-bit or chipkill errors. Knowledge about the increased likelihood of future multi-bit or chipkill errors could be used by an adaptive system to take proactive measures to protect against errors.

To shed some light on this question, we plot in Figure 3.11 (left) the probability that a node develops a multi-bit error after seeing previous errors of different types for BG/L and BG/P. More precisely, each set of 5 bars in the graph shows the following 5 probabilities. The first bar in each group of five bars

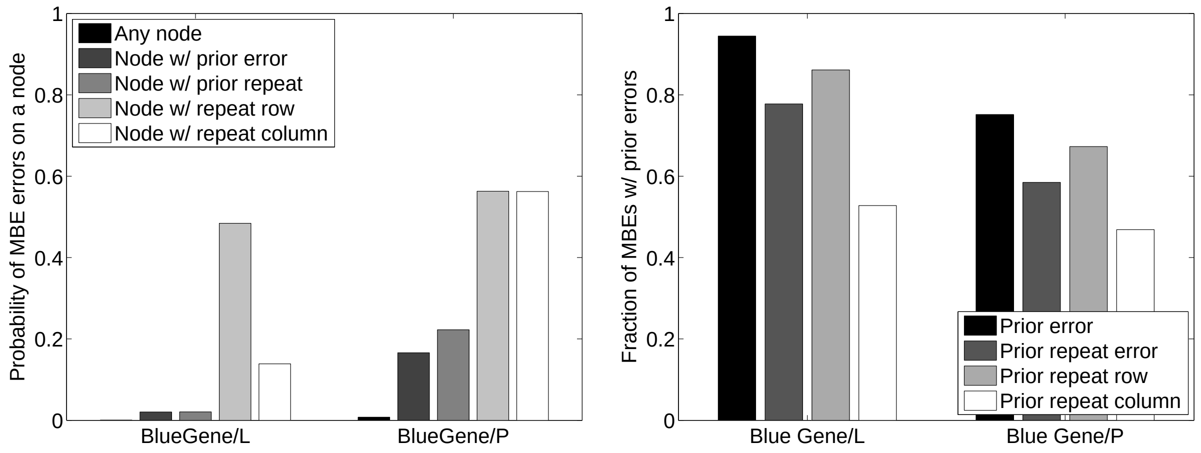


Figure 3.11: The relationship between multi-bit errors and prior errors.

represents the baseline probability of a random node seeing a multi-bit error. (Note, that this probability is so small that it is barely visible in the graph.) The second bar represents the probability that a node that has seen a prior error (of any type) will later experience a multi-bit error. The other three bars show the probability that a node will experience a later multi-bit error after experiencing a repeat address, a repeat row or a repeat column, respectively. The figure clearly indicates that for both systems the probability of a multi-bit error increases, after seeing previous errors. It also shows that the probability increases dramatically if a previous error was a repeat error.

Figure 3.11 (left) tells us only that the probability of multi-bit errors increases after other errors have been observed; it does not tell us whether most multi-bit errors were in fact preceded by prior errors (which a system could use as an early warning sign of impending multi-bit errors). In order to look at the latter, Figure 3.11 (right) plots the fraction of multi-bit errors that were preceded be the four types of errors we considered previously (any error, repeat address, repeat row, repeat column). The graph shows that multi-bit errors don't usually happen without prior warning: 60-80% of multi-bit errors were preceded by repeat addresses, 70-85% of multi-bit errors were preceded by a repeat row and 40-50% of multi-bit errors were preceded by a repeat column.

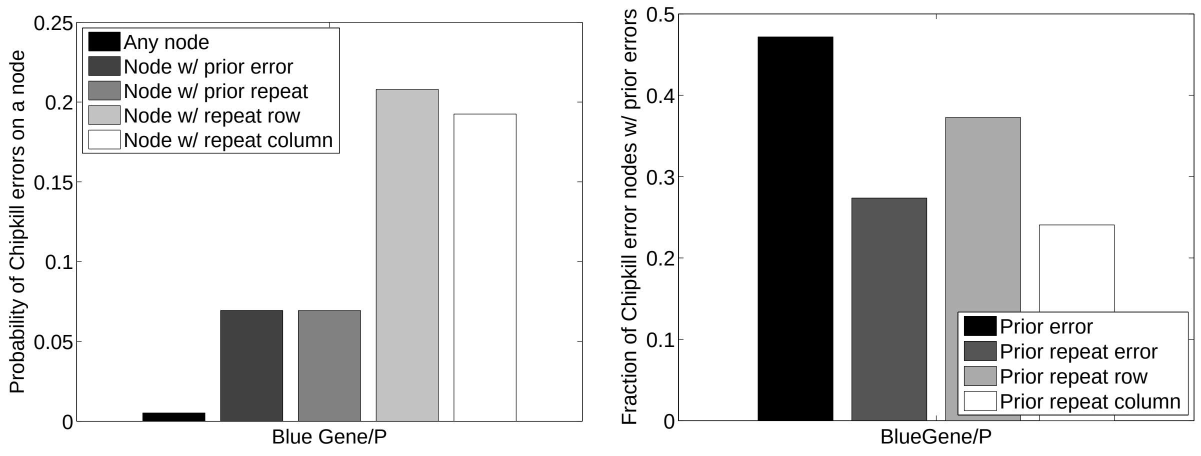


Figure 3.12: The relationship between chipkill errors and prior errors.

Figure 3.12 repeats the same analysis for chipkill errors, rather than multi-bit errors (for BG/P only, as chipkill errors do not apply to BG/L). While the overall probabilities are smaller (due to the lower rate of chipkill errors), we observe the same trends. Prior errors greatly increase the probability of a later chipkill error. Among nodes with prior error the probability increases to 7%. If there is a repeat row or repeat column present in the system, the likelihood of a later chipkill error increases to around 20%.

3.4 Implications for system design

An underlying theme throughout the previous section has been the study of hard errors as the dominating error mode among DRAM errors in the field. Compared to soft errors, hard errors have a greater potential to increase error rates, due to their repetitive nature, and to increase the chance of future uncorrectable errors. On the positive side, the repeating nature of hard errors makes them also more predictable than soft errors creating a potential for taking proactive measures against them. In this section, we discuss various implications on resilient system that follow from the insights derived from our measurement study.

3.4.1 Page retirement

While error protection at the hardware level in the form of ECC is effective, it is not always the most desirable option. In addition to the cost factor, another concern, in particular for the more powerful ECC codes, is the increase in energy consumption and the possible impact on performance.

As an alternative to more powerful ECC codes (such as chipkill), and an extra level of protection in addition to the use of ECC DIMMs, one could consider the retirement of pages that have experienced previous (hard) errors. Page retirement can be accomplished by having the OS isolate pages containing errors and prevent them from being allocated in the future. While this technique is not widely used in today's data centers, some operating systems, such as Solaris [142, 21], offer built-in support for page retirement. For the standard Linux kernel there exists a patch that allows one to retire bad pages [118]. However, there is no rigorous study on the possible effectiveness of page retirement for realistic error patterns and there is no published work comparing different policies for deciding on when to retire a page.

The main trade-off in retiring pages is the amount of memory lost due to retired pages versus the number of future errors prevented. An ideal retirement policy detects as early as possible pages that are likely to develop a large number of errors in the future and retires only those pages. We have observed several indicators that lend themselves well for such predictions. Below are a few simple policies that were directly derived from the findings in Section 3.3.

repeat-on-address: Once an address experiences a repeat error the corresponding page is retired.

1-error-on-page: Since a large fraction of addresses with errors experiences a repeat, this policy pessimistically assumes after the first occurrence of an error on a page that it will turn into a hard error and retire the page.

2-errors-on-page: This policy retires a page once two errors have been observed on this page (either on the same address or on two different addresses on the page).

repeat-on-row: Since a row with 2 unique addresses has high chances of developing additional errors, this policy retires all the pages on a row after two errors have been observed.

repeat-on-column: Equivalent to repeat-on-row, but for columns.

We simulate all of the above policies on our trace data for BG/L and BG/P and report the results in Table 3.5. For each policy we include the average number of pages it retires per machine with errors, the 95th percentile of the number of pages retired per machine, the percentage of all errors in the system that would have been prevented by this policy (because they fall on previously retired pages) and the percentage of all multi-bit and chipkill errors that could have been prevented.

		All nodes with errors					Nodes w/ multi-bit/ chipkill error	
System	Policy	Pages retired	95%ile pages retired	Errors avoided (%)	MBEs avoided (%)	Chipkill avoided (%)	Pages retired	95%ile pages retired
BG/L	repeat on address	2.2	2	94.2	88.1	N/A	15.8	103.25
	any 1 error / page	3.8	4	96.8	96.7	N/A	42.4	319
	any 2 errors / page	2.4	3	94.9	94.8	N/A	24.6	123.8
	repeat on row	33.9	32	95.6	97.3	N/A	245.5	1620
	repeat on column	14,336	16,384	96.5	90.6	N/A	257,930	1,212,416
BG/P	repeat on address	4.8	18	86.3	86.4	61.8	4.8	18
	any 1 error / page	17.6	62.7	91.4	91.5	71.0	17.7	62.7
	any 2 errors / page	6.9	25.6	88.1	88.1	64.7	6.9	25.6
	repeat on row	158.0	576	92.6	92.7	77.0	158	576
	repeat on column	49,989	266,650	91.9	92	67.3	49,972	266,650

Table 3.5: Effectiveness of page retirement

We find that even the simple policies we are considering are quite effective at reducing the number of errors a system would observe: All policies are able to prevent nearly 90% of all errors. The most aggressive policies (retiring a page immediately after just one error, or retiring whole rows and columns) are able to avoid up to 96% of all errors. The main difference between policies lies in the cost involved in achieving this performance. The number of pages retired per machine averages at only 2.2 - 4.8 for the repeat-on-address policy, which is a small price to pay for a large gain in the number of avoided errors. For policies that retire entire rows or columns this number can grow into hundreds or thousands of pages retired per machine. Retiring entire columns is particularly expensive, due to the large number of pages that a column spans, and is prohibitive, at least in the form of the very simple policies that we have experimented with.

Another interesting finding from our simulation study is the effectiveness of page retirements in avoiding multi-bit and chipkill errors. All policies are able to avoid around two thirds of all chipkill errors and nearly 90% of all multi-bit errors. While a system with chipkill ECC would have been able to mask all of these errors, the high reduction of errors under page retirement is still interesting as it does not come with an increase in hardware cost or energy consumption. The only price to pay is the reduced amount of memory available, due to retired pages.

While the average number of pages retired per machine averaged across all machines with errors is low when the right policy is chosen, this number might be higher for machines that experience multi-bit or chipkill errors (or more precisely would have experienced multi-bit or chipkill errors in the absence of page retirement). We therefore also computed the statistics for the number of pages retired per machine for only those machines in our dataset that experienced multi-bit and chipkill errors and report the results in the right half of the above table. We find that both the average number of pages retired and the 95th percentile of the number of pages retired is still very small, compared to the total amount of memory in modern server systems. For example, under the repeat-on-address policy 5-16 pages are retired for an average machine with errors. A machine in the 95th percentile of number of retired pages, still sacrifices only 18-104 pages, i.e. less than half a MByte of total DRAM space. Even the for the more aggressive 1-error-on-a-page policy the number of retired pages is still in the same order of magnitude. More elaborate techniques based on statistical modeling or machine learning might be able to further improve on the cost-efficiency trade-off of page retirement policies.

3.4.2 Selective error protection

Several of our findings indicate that errors are not uniformly distributed in space. For example, we saw evidence that some parts of a chip and of the physical address space experience higher error rates than others. This implies that selective error protection mechanisms might be an interesting avenue for future work. For example, approaches along the lines of the work on virtualized and flexible ECC [161] might provide effective solutions that exploit the differences in error rates in different parts of the system.

3.4.3 Proactive error detection and monitoring

The two previous subsections provide examples for techniques that operating systems can use to exploit the characteristics of hard errors in order to reduce the negative impact of DRAM errors on system availability. However, such techniques require that the operating system has full knowledge of all errors happening at the underlying hardware level, including error counts, as well as more detailed information, such as the addresses that were affected. ECC protection used in most server systems masks the presence of errors and it is typically not trivial to obtain location information on errors. Our findings about the nature of DRAM errors provide strong encouragement to improve error tracking and reporting to the operating systems.

The second factor limiting the amount of knowledge about the underlying error process stems from the fact that DRAM errors are latent, i.e. they will not be detected until the affected cell is accessed. The chances that an error will eventually lead to an uncorrectable error, causing system downtime, increases the longer it is left latent. While hardware scrubbers provide an attempt to proactively detect errors and hence reduce this time, we observed in Section 3.3.3 that their effectiveness might be limited. We speculate that this is likely due to the passive monitoring approach that they are taking, rather than actively attempting to expose errors. Given the large amount of idle time that typical servers in data centers experience [10], it might be worthwhile to invest a small fraction of this idle time to periodically run a memory test, similar in functionality to memtest86, that actively probes the DRAM for memory errors. Memory tests have a much higher potential for detecting hard errors than a scrubber, since they can create access patterns that stress the memory, and because they actively write to memory cells, rather than just checking the validity of currently written values. Such tests could either be done periodically on all machines, or they could be used selectively in cases where there is a suspected hard error, e.g. after observing an earlier error in the system (that has not yet repeated).

3.4.4 Effectiveness of hardware mechanisms

While this was not the original focus of our study, our analysis lets us also draw conclusions about the practical value of hardware mechanisms, such as chipkill ECC, in reducing the rate of machine crashes due to uncorrectable errors. In the presence of only soft errors, the occurrence of error patterns requiring chipkill ECC would be extremely unlikely. Our observation that a large number of errors observed in the field is likely due to hard errors provides firmer grounding for using these techniques in practice, despite their added cost and energy overheads.

Only one earlier study [126] that is based on large-scale field data comments on the effectiveness of chipkill, however they only observe that in their systems under study hardware platforms with chipkill show lower rates of uncorrectable errors than hardware platforms without chipkill. They are not able to quantify how much of this difference in error rates is due to the use of chipkill versus other hardware differences between the platforms.

Our fine-grained data allowed us to quantify exactly the number of errors that required chipkill to be corrected and that would have led to a machine crash in the absence of chipkill. We find that a significant fraction of machines experienced chipkill errors, i.e. errors whose correction was only possible with the use of chipkill techniques. In fact, among the errors in our study, a large fraction (17%) of them required the use of chipkill for correction providing some tangible benefits of the use of chipkill. We can therefore conclude that for systems with stringent availability requirements the reduction in machine crashes due to uncorrectable errors might make chipkill well worth the price.

3.4.5 System evaluation

Any evaluation of the impact of DRAM errors on system reliability or the effectiveness of mechanisms protecting against them relies on realistic assumptions about the characteristics of the underlying error process. In the absence of field data (or realistic models built based on field data), both analytical and experimental work typically rely on very simplistic assumptions about errors. For example, analytical models often assume that errors follow a Markov process and experimental work often relies on injecting errors at uniformly randomly generated locations. Given the high occurrence rate of hard errors, these simple approaches are likely to give misleading results (or results that represent only the less relevant scenario of a system experiencing only soft errors), as they do not capture any of the correlations and patterns present in hard errors.

While we are hoping that the findings we report here will help researchers and practitioners to base their work on more realistic assumptions on DRAM errors, we believe that more work in this direction is necessary. Towards this end, we are currently working on developing statistical models capturing the various properties of DRAM error process that can be used to generate realistic patterns in simulation or for error injection.

3.5 Conclusions

While a large body of work has been dedicated to studying the characteristics of DRAM errors and how to best protect against them, the large majority of this work has focused on soft errors in DRAM. Our work presents the first study based on data from a large number of production systems that shows that a large fraction of errors observed in the field can be traced back to hard errors. For all systems we studied, more than a third of all memory banks that experienced errors show signs of hard errors, most commonly in the form of repeating errors on the same physical address within less than 2 weeks. Repeating errors on the same row/column are also common error modes. For some systems, as many as 95% of all observed errors can be attributed to hard errors. We also provide a detailed study of the statistical characteristics of hard errors. Some of these provide direct insights useful for protecting against errors. For example, we observe that not all areas in memory are equally likely to be affected by errors; specific regions such as low rows/columns have higher error probabilities. We speculate that this might be due to different usage patterns in different memory areas, as we observe for example that those areas used by the OS tend to see larger error counts. Furthermore, from the perspective of the OS, a large fraction of the errors observed in a system is usually concentrated on a small set of pages providing some motivation for proactively retiring pages after they experience errors. We also observed that errors that have the highest potential to be uncorrectable, such as multi-bit errors and errors that require chipkill for correction, are usually preceded by more benign early warning signs, such as repeating errors on individual addresses, rows or columns. Finally, we observe that a significant number of errors is complex enough to require more than simple SEC-DED error correction to be corrected. A significant number of nodes with correctable errors in our study activated more advanced ECC mechanisms (20%-45% activated redundant bit-steering, and 15% activated Chipkill) and a large fraction (17%) of all errors required the use of chipkill for error correction.

As a second contribution, we identify various implications on resilient system design that follow from the insights derived from our measurement study. One of our findings is that simple page retirement policies can potentially mask a large number of errors with only a small sacrifice in the amount of available DRAM. For example, a simple policy that retires a page after the first repeat error on an address on this page can mask up to 95% of all errors and up to 60% of errors that would require chipkill for correction, while giving up only a few dozen pages of main memory. This is an interesting finding, since based on discussions with administrators of large datacenters, the use of page retirement is not widely spread in practice, although it has been implemented in some systems in the past [21]. On the other hand, we find that a commonly used technique for proactively detecting memory errors, the use of background memory scrubbers, might not be as effective as one might think. We hypothesize that this is because a large fraction of errors are intermittent, i.e. they manifest only under certain access patterns. This observation, together with the observed high frequency of hard (and hence repeatable) errors, might make it worthwhile to use the idle time that most servers in datacenters experience to periodically run a memory test to actively probe for errors, in particular after observing prior errors on a node. Finally, the fact that different areas of memory experience different error rates and that usage likely plays a role in error frequencies suggests an interesting avenue for future work might be selective error protection mechanisms, where different protection mechanisms are used for different areas of memory.

Chapter 4

Impact of Temperature on the Storage Stack

4.1 Introduction

Data centers have developed into major energy hogs. The world's data centers are estimated to consume power equivalent to about seventeen 1,000 MW power plants, equaling more than 1% of total world electricity consumption, and to emit as much carbon dioxide as all of Argentina [68]. More than a third, sometimes up to one half of a data center's electricity bill is made up by electricity for cooling [11, 77]. For instance, for a data center consisting of 30,000 square feet and consuming 10MW, the yearly cost of running the cooling infrastructure can reach up to $4-8 million [110].

Not surprisingly, a large body of research has been devoted to reducing cooling cost. Approaches that have been investigated include, for example, methods to minimize air flow inefficiencies [110, 139], load balancing and the incorporation of temperature awareness into workload placement in data centers [14, 112, 117, 128], and power reduction features in individual servers [44, 47].

Interestingly, one key aspect in the thermal management of a data center is still not very well understood: controlling the setpoint temperature at which to run a data center's cooling system. Data centers typically operate in a temperature range between 20C and 22C, some are as cold as 13C degrees [15, 120]. Due to a lack of scientific data, these values are often chosen based on equipment manufacturers' (conservative) suggestions. Some estimate that increasing the setpoint temperature by just one degree can reduce energy consumption by 2 to 5 percent [15, 18]. Microsoft reports that raising the temperature by two to four degrees in one of its Silicon Valley data centers saved $250,000 in annual energy costs [120]. Google and Facebook have also been considering increasing the temperature in their data centers [120].

While increasing data center temperatures might seem like an easy way to save energy and reduce carbon emissions, it comes with some concerns, the most obvious being its impact on system reliability. Unfortunately, the details of how increased data center temperatures will affect hardware reliability are not well understood and existing evidence is contradicting. To address this problem, the first half of our Sigmetrics 2012 paper [40] and its associated technical report [39] provide a detailed study of the effects of temperature on hardware reliability by analyzing a large amount of field data. The data comes from three different organizations spanning several dozen data centers and covers a diverse set of common reliability issues, including hard disk failures, latent sector errors in hard disks, uncorrectable errors in

DRAM, DRAM replacements, and general node outages.

The focus of this chapter is another important concern that arises with increasing data center temperatures: the effect on server performance and power consumption. Hard disks, memory, and CPUs all employ a number of hardware reliability mechanisms and features intended to maintain the integrity of data at higher temperatures, and protect the hardware against damage or excessive errors. These reliability mechanisms and features all impose performance penalties for the components in question. In addition, increased temperatures also have a marked effect on server energy consumption, as they will lead to increases in power leakage and higher (server internal) fan speeds.

This chapter contains an experimental study using a testbed that includes a thermal chamber and a large set of different workloads to better understand the effects that temperature has on the performance and power usage of systems. We provide extensive experimental results across a wide array of representative workloads that quantify the range of performance and power penalties across the entire spectrum of configuration options. This study is intended as a guide to data center operators in understanding the tradeoffs between operating at higher temperatures and the inherent performance and power penalties of doing so.

4.2 Temperature and performance

While it is widely known that higher temperatures might negatively affect the reliability and lifetime of hardware devices, less attention is paid to the fact that high temperatures can also negatively affect the *performance* of systems. For example, in order to protect themselves against a possibly increasing rate of Latent Sector Errors (LSEs), some hard disk models enable Read-after-Write (RaW) when a certain temperature threshold is reached. Under RaW, every write to the disk is converted to a Write Verify command, or a Write followed by a Verify operation, reading the sector that has just been written and verifying its contents [140, 141] [1]. Also, when CPU and memory temperatures reach a certain threshold, most high-end servers employ CPU throttling (dynamic voltage frequency scaling) and memory throttling (of the memory bus).

Unfortunately, features such as RaW are often considered trade secrets and are not well documented. In fact, even within a company manufacturing hardware those features and associated parameters are regarded confidential and not shared outside product groups. The goal in this part of our work is to investigate experimentally how the performance of different components changes with increasing temperatures.

4.3 Experimental Setup

To study the performance of a server under increasing ambient temperatures, we set up a testbed using a thermal chamber. The thermal chamber is large enough to fit an entire server inside it, and allows us to exactly control temperature within a range of -10C to 60C with a precision of 0.1C. How ambient temperature affects the temperature of server-internal components depends on many factors, including the design of the cooling system and the server and rack architecture. Therefore, instead of directly

[1] Note that Write Verify is not specified in the ATA standard, which might explain the absence of a performance hit for most SATA drives, in the following sections.

predicting the impact of data center ambient temperature on a system, we present our results as a function of the temperature of server internal components.

The server we use in our study is a Dell PowerEdge R710, a model that is commonly used in data center server racks. The server has a quad-core 2.26 GHz Intel Xeon 5520 with 8MB L3, with 16GB DDR3 ECC memory, running Ubuntu 10.04 Server with the 2.6.32-28-server Linux kernel. We also equipped the server with a large variety of different hard disk drives, including both SAS and SATA drives and covering all major manufacturers:

Manufacturer	Model	Interface	Capacity	RPM
Hitachi	Deskstar	SATA	750GB	7200
Western Digital	Caviar	SATA	160GB	7200
Seagate	Barracuda	SATA	1TB	7200
Seagate	Constellation	SAS	500GB	7200
Seagate	Cheetah	SAS	73GB	15000
Fujitsu	MAX3073RC	SAS	73GB	15000
Hitachi	Ultrastar	SAS	300GB	15000

We use a wide range of workloads in our experiments, including a set of synthetic microbenchmarks designed to stress different parts of the system, and a set of macrobenchmarks aiming to model a number of real world applications:

STREAM: A microbenchmark measuring bandwidth of sequential memory accesses [86]. We used an implementation from the lmbench suite [88, 135] and benchmarked the performance of accessing 4GB of memory.

GUPS: Microbenchmark that measures memory random accesses, in giga-updates-per-second, as defined by the High Performance Computing Challenge [114]. We tested the performance of 8KB-chunk updates randomly to 4GB of memory.

Dhrystone: A well-known microbenchmark that evaluates the CPU performance for integer operations [157].

Whetstone: A well-known CPU benchmark for floating-point performance [31]. Our implementations of *Dhrystone* and *Whetstone* were obtained from the Unixbench suite [103].

Random-Read/Write: A synthetic workload comprised of independent 64KB read (or write) requests issued back-to-back at random disk sectors.

Sequential-Read/Write: Since a pure sequential workload would stress the on-disk cache, we opt for a synthetic workload with a high degree of sequentiality, instead. We pick a random disk sector, and issue back-to-back 64KB read (or write) requests on consecutive sectors for 8MB following the initial request.

OLTP-Mem: We configured TPC-C [150], a commonly used database benchmark modeling on-line transaction processing (OLTP), with 30 warehouses resulting in a 3GB *memory-resident* database.

OLTP-Disk: Models the same workload as *OLTP-Mem*. To make the workload I/O-bound, we configured the database with 70 warehouses (7GB), using 4GB RAM.

DSS-Mem: We configured TPC-H [151], a commonly used database benchmark modeling decision support workloads (DSS), with a 1GB memory-resident MySQL InnoDB database.

DSS-Disk: Another TPC-H based workload, this time configured with a database of 10GB and a 3.4GB buffer pool, resulting in a disk-bound workload.

PostMark: A common file system benchmark [69], which we configured to generate $50-5000$KB files, and modified it to avoid using the OS cache entirely, so that all transactions are directed to disk.

BLAST: An application [3] used by computational biology researchers, acting as a high-performance computing benchmark that stresses both the CPU and memory. We used the parallel *mpiBLAST* implementation [32] and ran 10 representative queries on a 5GB library.

4.4 Temperature and disk performance

To study the effect of temperature on disk performance, we ran our disk-bound workloads against each of the drives in our testbed, while placing the drive in the heat chamber and gradually increasing the temperature inside the chamber. The two graphs in Figure 4.1 show the results for the random-read and random-write microbenchmarks, as a function of the drive internal temperatures, as reported by the drives' SMART statistics. Results for sequential-read and sequential-write were similar. We observe that all SAS drives and one SATA drive (the Hitachi Deskstar) experience some drop in throughput for high temperatures. The drop in throughput is usually in the 5-10% range, but can be as high as 30%. Because of the fact that the throughput drop for a drive happens consistently at the same temperature, rather than randomly or gradually, and that none of the drives reported any errors, we speculate that it is due to protective mechanisms enabled by the drive. For example, in the case of the write workloads (which show a more significant drop in throughput) this drop in throughput might be due to the enabling of features such as RaW.

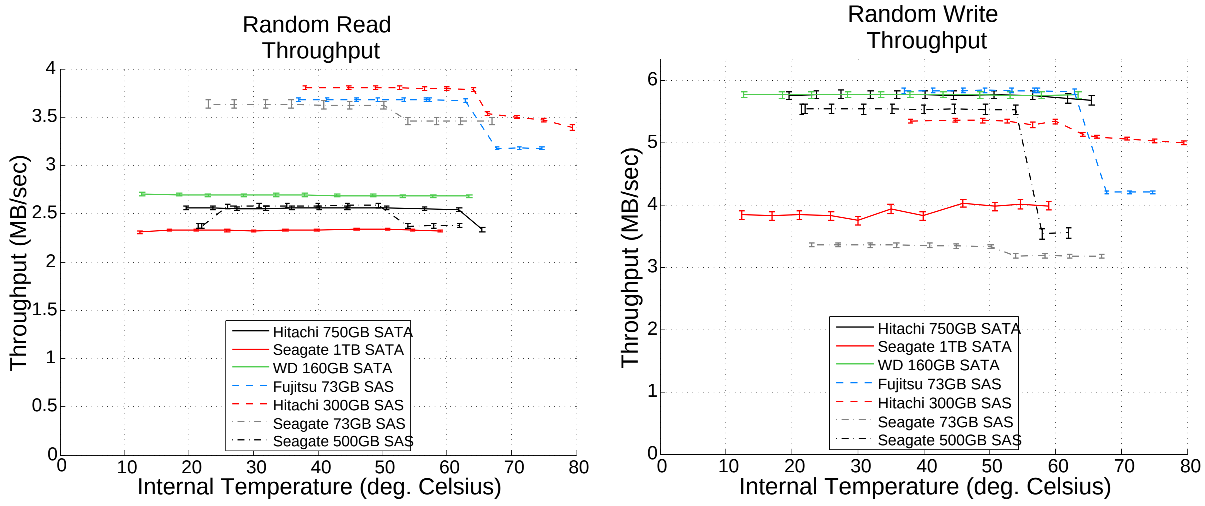


Figure 4.1: Disk throughput under a synthetic random read and random write workload, respectively, as a function of disk internal temperature. Results for sequential read and sequential write workloads were comparable.

An interesting question is: at what temperature does the throughput start to drop? We observe in Figure 4.1 drops at either around 50C (for the Seagate SAS drives) or 60C (for the Fujitsu and Hitachi SAS drives). However, these are disk internal temperatures.

The two graphs in Figure 4.2 translate ambient temperatures (inside the heat chamber) to the observed drives' internal temperatures. Note that the 15,000 RPM drives naturally run at a hotter internal temperature. The markers along the lines mark the points where we observed a drop in throughput. We observe a drop in throughput for temperatures as low as 40C (for the Seagate 73GB and Hitachi SAS drives), 45C for the Fujitsu and Seagate 500GB SAS drives, and 55C for the Hitachi Deskstar, ranges

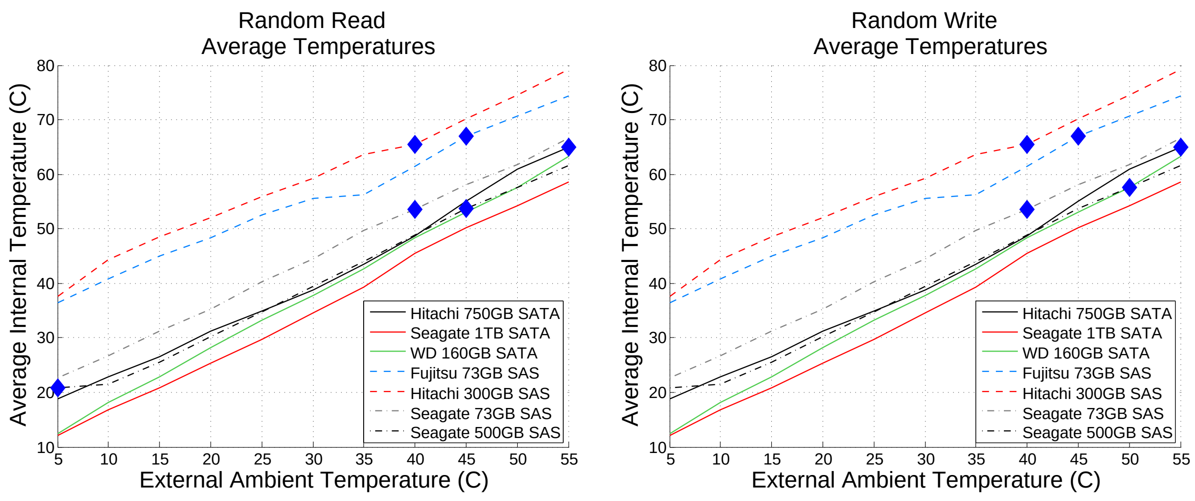


Figure 4.2: Disk internal temperature as a function of ambient temperature for different drive models and random reads (left) and random writes (right).

that are significantly lower than the maximum of 50-60C that manufacturers typically rate hard disks for.

While data centers will rarely run at an average inlet temperature of 40C or above, most data centers have hot spots which are significantly hotter than the rest of the data center, and which might routinely reach such temperatures.

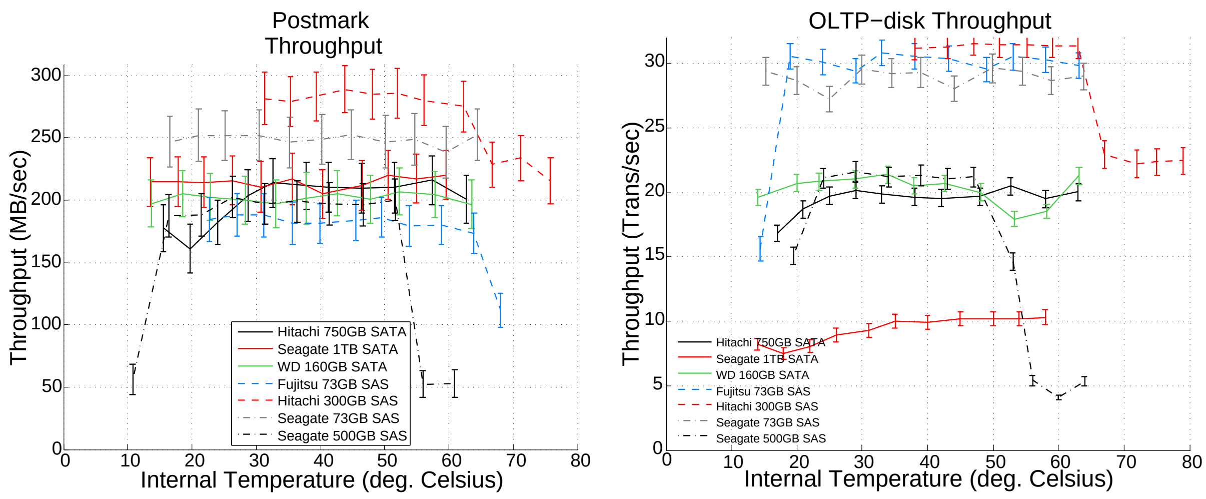
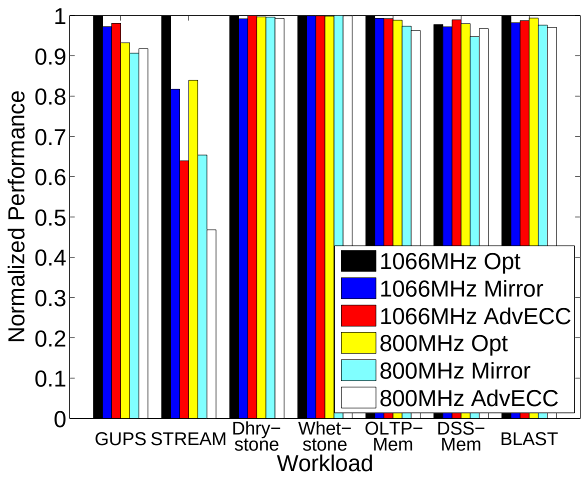


Figure 4.3: Throughput under two different I/O intensive workloads (Postmark, OLTP-disk) as a function of disk internal temperature.

Figure 4.3 shows how temperature affects the throughput of two of our disk-intensive applications, Postmark and OLTP-disk. We observe similar trends as for the microbenchmarks, with throughput drops at the same temperature point. Interestingly, the order of magnitude in the throughput drop for Postmark and OLTP-disk is in most cases even larger than for the synthetic microbenchmarks. The drop in throughput for Hitachi and Seagate SAS drives increases to 10-20%, while the throughput drop for Fujitsu and Seagate SAS drives is in the range of 40% and 80%, respectively. While the performance impact on DSS-disk is somewhat lower, it is important to note that this is the average drop in performance across all queries in the DSS workload. For each SAS drive there are some queries that are impacted by as much as 10–25%. The drops observed for DSS-disk looked more similar in magnitude to those for the synthetic benchmarks.

4.5 Temperature and CPU/memory performance

Most enterprise class servers support features to protect the CPU and memory subsystem from damage or excessive errors due to high temperatures. These include scaling of the CPU frequency, reducing the speed of the memory bus, and employing advanced error correcting codes (ECC) for DRAM. For example, our server supports a continuous range of CPU frequencies, bus speeds of either 800MHz or 1066MHz, and three memory protection schemes: single-error-correction and double-error-detection (SEC-DED), advanced ECC (AdvEcc), which allows the detection and correction of multi-bit errors, and mirroring, which provides complete redundancy. Server manuals tend to be purposely vague as to when such features are enabled (CPU and memory bus throttling can be automatically activated by the server), or possible performance impact. In particular, for the memory options it is difficult to predict how they affect performance and power consumption. Since running data centers at higher temperatures might necessitate the use of such features more frequently, we use our testbed to study their impact on performance and power consumption.



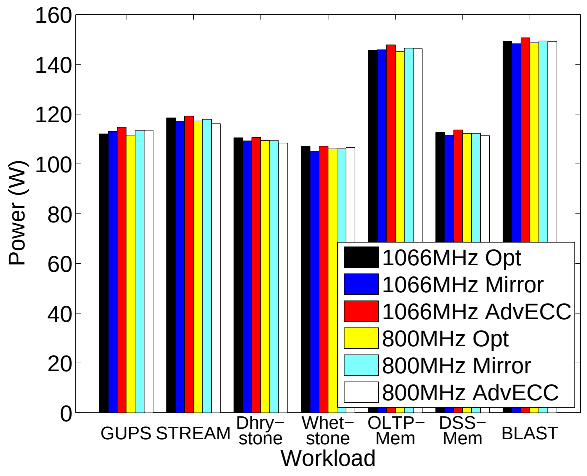


Figure 4.4: The effect of memory error protection and bus speed on performance (left) and power consumption (right).

For the temperature range we experimented with (heat chamber temperatures up to 55C, significantly higher than the 35C inlet temperature at which most servers are rated) we did not observe any throttling triggered by the server. To study the effect of different memory features, we manually configured the server to run with different combinations of memory bus speed (800MHz vs. 1066MHz) and ECC schemes (SEC-DED, AdvEcc, Mirror). The effect on throughput for the different benchmarks is shown in Figure 4.4 (left). Throughput is normalized by the maximum attainable throughput, i.e. the throughput achieved when combining a 1066MHz bus speed with the SEC-DED ECC scheme. The results for the two microbenchmarks designed to stress the memory (GUPS and Stream) show that drops in throughput can potentially be huge. Switching to the lower bus speed can lead to a 20% reduction in throughput.

The effect of the ECC scheme is even bigger: enabling AdvECC can cost 40% in throughput. The combination of features can cause a drop of more than 50%. For the macrobenchmarks modeling real-world applications the difference in throughput is (not surprisingly) not quite as large, but can reach significant levels at 3–4%. We also measured the server's power consumption (Figure 4.4 (right)), and found that the impact of memory configurations on server power is small (1-3%) compared to increases of up to 50% due to increased temperatures that we will observe in the next section.

4.6 Increased server energy consumption

Increasing the air intake temperature of IT equipment can have an impact on the equipment's power dissipation. Many IT manufacturers start to increase the speed of internal cooling fans once inlet air temperatures reach a certain threshold to offset the increased ambient air temperature. Also, leakage power of a processor increases with higher temperatures, and can make up a significant fraction of a processor's total power consumption. To study the effect of increasing ambient temperatures on a server's power consumption, we repeated all our earlier experiments with a power meter attached to our server and, in addition, monitored fan speeds.

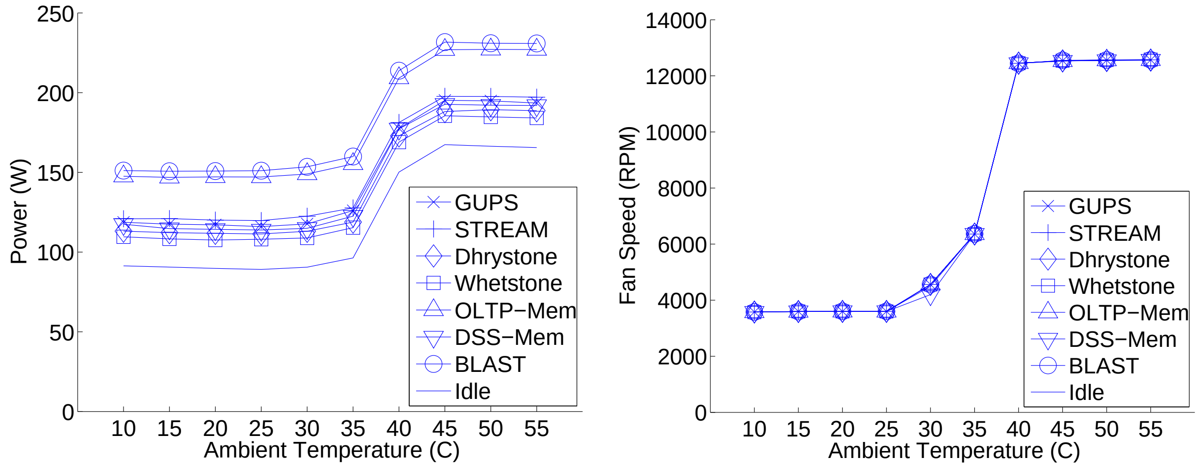


Figure 4.5: The effect of ambient temperature on power consumption (left) and server fan speeds (right).

Figure 4.5 (left) shows the server's power usage as a function of the ambient (thermal chamber) temperature for the CPU and memory intensive workloads. While the absolute energy used by different workloads varies widely, we observe the same basic trend for all workloads: power consumption stays constant up to 30C and then begins to continually increase, until it levels off at 40C. The increase in power consumption is quite dramatic: up to 50%.

An interesting question is whether this increase in power comes from an increase in fan speed (something that can be controlled by the server) or from increased leakage power (which is governed by physical laws). Unfortunately, it is not possible to measure leakage power directly. Nevertheless, there is strong evidence that the increase in power is dominated by fan power: Figure 4.5 (right) plots the fan speed as a function of the ambient temperature for all workload experiments. We observe that the temperature thresholds we notice for which fan speeds increase line up exactly with the temperatures at which when power consumption increases. We also observe that power consumption levels off once fan speeds level off, while leakage power would continue to grow with rising temperatures. As a result, we can conclude that as ambient temperature increases, the resulting increase in power is significant and can be mostly attributed to fan power. By comparison, leakage power is negligible.

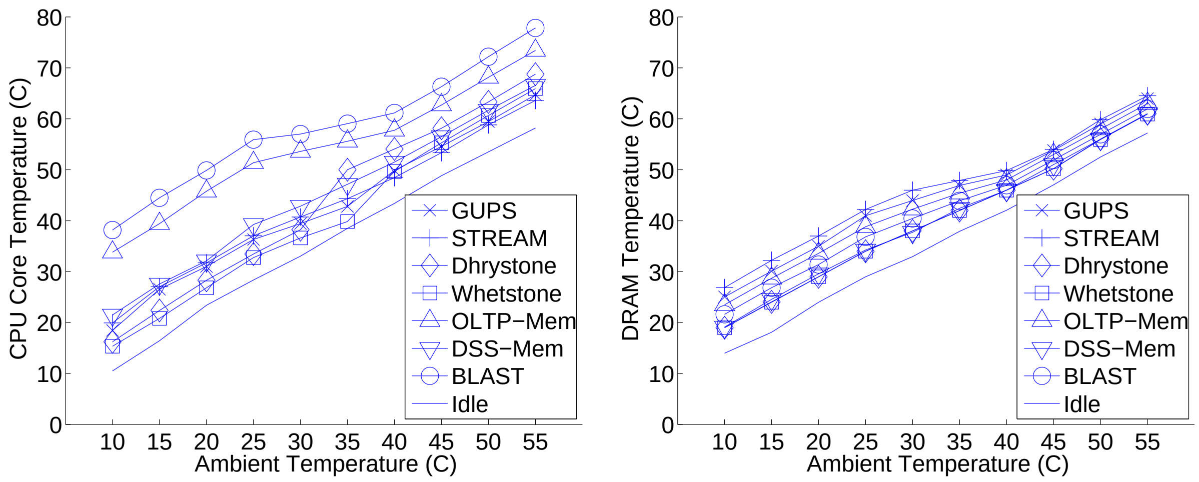


Figure 4.6: The effect of ambient temperature on CPU temperature (left) and memory temperature (right).

Another interesting observation is that power usage starts to increase at the same ambient temperature point for all workloads, although server internal temperatures vary widely across workloads, which means fan speeds increase based on ambient rather than internal temperature. Figure 4.6 shows the CPU and memory temperature as a function of ambient temperature for the different workloads, and for an idle server. We see, for example, that CPU core temperature is more than 20C higher for BLAST and OLTP-Mem than for most other workloads. That means that for many workloads the server internal temperatures are still quite low (less than 40C) when the fan speeds start to increase. In particular, we observe that for an idle server, the temperature measured at the CPU and memory is still at a very modest 25-30C [2] when the fan speeds start to increase. This is an important observation, since most servers in data centers spend a large fraction of their time idle. As such, we conclude that smart control of server fan speeds is imperative to running data centers hotter. A significant fraction of the observed increase in power dissipation in our experiments could likely be avoided by more sophisticated algorithms controlling the fan speeds.

4.7 Reduced safety margins

One concern with increasing data center temperatures is that most data centers tend to have hot spots that are significantly hotter than the average temperature in the facility. When raising the temperature setpoint in a data center's cooling system, it is important to also keep in mind how this will affect the hottest part of the system, rather than just the system average. In addition to hot spots, another concern is reduced safety margins: most servers are configured with a critical temperature threshold and will shut down when that threshold is reached, in order to avoid serious equipment damage. As the ambient temperature in a data center increases, equipment will be operating closer to the maximum temperature, reducing the time available to shut down a server cleanly or take protective measures in

[2]For reference, DRAM, for example, is typically rated for up to 95C.

the case of data center events, such as cooling or fan failures.

To better understand temperature imbalances we studied data collected from January 2007 to May 2009 at 6 different data centers (DCs) at Google covering three different disk models. For each of the disks, we have monthly reports of the average (internal) disk temperature and temperature variance in that month. The data was collected and analyzed using the Systems Health infrastructure at Google [113]. The infrastructure consists of 3 layers: data collection, a distributed storage repository, and an analysis framework. The data collection was done by polling the disks' internal self-monitoring facility (SMART). The data was stored in Bigtable [22], where different columns represent different variables, different rows represent different machines, and different versions are used to keep a time-ordered history of variable values [113]. Data analysis was done using a Mapreduce job written in the Sawzall language for data extraction [113], and R for statistical analyisis and graph generation. The table below summarizes our data:

Model ID	#DCs	#Disks	#Disk Months
3	3	18,692	300,000
4	3	17,515	300,000
6	4	36,671	300,000

In addition, Lawrence Livermore National Laboratories (LANL) have made available data on node outages for more than 20 of their high-performance computing clusters, including information on the root cause of an outage and the duration of the outage. The data can be downloaded from the Usenix Computer Failure Data Repository [153] and a more detailed description of the data and systems can be found in [124]. For one of LANL's clusters, periodic temperature measurements from a motherboard sensor are also available. We refer to this system as LANL-system-20, since the ID for this system on LANL's web page is 20.

We consider how much hotter the disk/node in the 95th and 99th percentile of the distribution in the data center is, compared to the median disk/node.

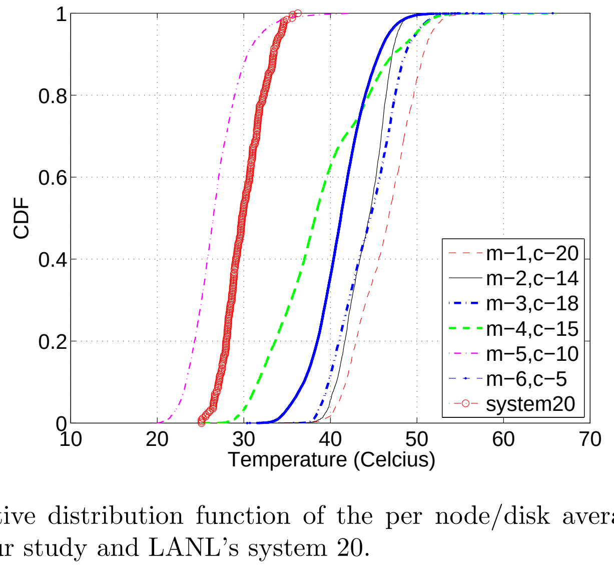


Figure 4.7: The cumulative distribution function of the per node/disk average temperatures for the Google data centers in our study and LANL's system 20.

Interestingly, the trends for temperature imbalances are very similar across data centers, despite the fact that they have been designed and managed by independent entities. We find that for all of Google's data centers in our study, and LANL's system 20, the node/disk in the 95th percentile is typically around 5 degrees C hotter than the median node/disk, and that the 99th percentile is around 8–10 degrees hotter than the median node/disk. Figure 4.7 shows the full CDFs of the per node/disk distribution for both the Google data centers and LANL's system 20.

4.8 Summary and Implications

Increasing data center temperatures creates the potential for large energy savings and reductions in carbon emissions. Unfortunately, the pitfalls possibly associated with increased data center temperatures are not very well understood, and as a result most data centers operate at very conservative, low temperatures. The experiments in this chapter quantify the range of performance and power penalties incurred when operating at higher temperatures. Indeed, both the disk and the CPU/memory subsystems can incur significant performance penalties due to reliability mechanisms and features that attempt to maintain the integrity of the data and protect the hardware against damage. These mechanisms tend to kick in only at very high temperatures, and the performance penalties can be significant, depending on the workload.

Our encouraging results on the impact of temperature on hardware reliability [40, 39] move the focus to other potential issues with increasing data center temperatures. One such issue is an increase in the power consumption of individual servers as inlet air temperatures go up. The two most commonly cited reasons for such an increase are increased power leakage in the processor and increased (server internal) fan speeds. Our experimental results show that power leakage seems to be negligible compared to the effect of server fans. In fact, we find that even for relatively low ambient temperatures (on the orders that are commonly found in the hotter areas of an otherwise cool data center) fan power consumption makes up a significant fraction of total energy consumption. Much of this energy might be spent unnecessarily, due to poorly designed algorithms for controlling fan speed.

4.9 Conclusion

We now conclude the part of the thesis focused on understanding and improving system reliability and temperature-related performance and power considerations in data centers. The following chapters will focus on building a programmable and controlable software storage stack, that allows vertical specialization and customization for data center applications.

Chapter 5

Software-Defined Caching

5.1 Introduction

An increasing number of enterprise applications have migrated to hosted platforms in private enterprise and public cloud data centers. Such platforms are typically virtualized, i.e., tenants deploy applications in virtual machines (VMs) whose access to the underlying resources (memory, storage, network) is shared with other tenants, and mediated by hypervisors such as Hyper-V, VMware ESX, or Xen. Uninhibited sharing of such resources in a multi-tenant environment leads to poor and variable application performance. While recent efforts give providers control over how resources like network [54, 8, 130, 115, 65] and storage [52, 131, 147, 53, 13] are shared, there is no coordinated end-to-end control of the distributed caching infrastructure, made up of storage caches at multiple places along the IO stack (inside VMs, hypervisors, storage servers; see Figure 5.1).

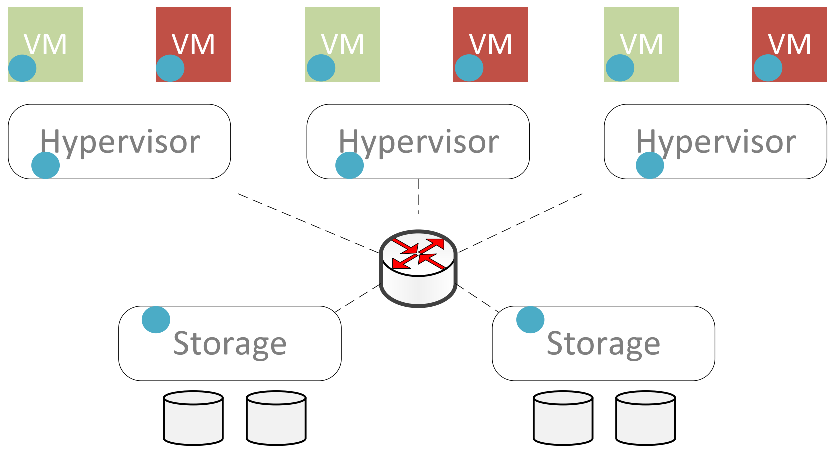


Figure 5.1: Simplified IO stack in a multi-tenant data center. Two tenants, a green and red one are shown, with 3 VMs each spread over 3 hypervisors. The circles represent typical caches on the IO stack.

Today, storage caches along the IO stack are transparent to both applications and cloud providers, lack workload-aware mechanisms, and are each managed in isolation, leading to multiple problems:

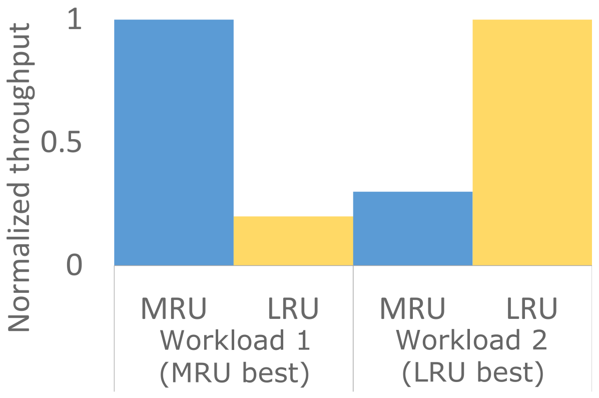
• **Lack of performance isolation.** Since caches are not tenant- or workload-aware, applications with different IO patterns and request rates sharing the same cache will impact each other's cache performance. For example, depending on the cache eviction policy, one application's large sequential reads can blast away another workload's working set. Even with scan-resistant cache management policies, such as ARC [89], aggressive clients with higher request rates will still be allocated larger portions of the cache.

• **Lack of customization.** Since caches are not tenant-aware, the entire cache is treated as a single pool with one cache write policy (write-through, write-back, etc.), despite different durability requirements of different applications, and one eviction policy, despite the fact that different workloads benefit from different cache eviction policies. For example, Figure 5.2a shows two IOMeter workloads under two different eviction policies, LRU and MRU [27] respectively. The workload on the left performs at its peak with an MRU policy, while the one on the right performs best with LRU. Today, if both workloads were running atop the same hypervisor, they would have to follow the same eviction policy, leading to performance penalties on the order of 4-5x.

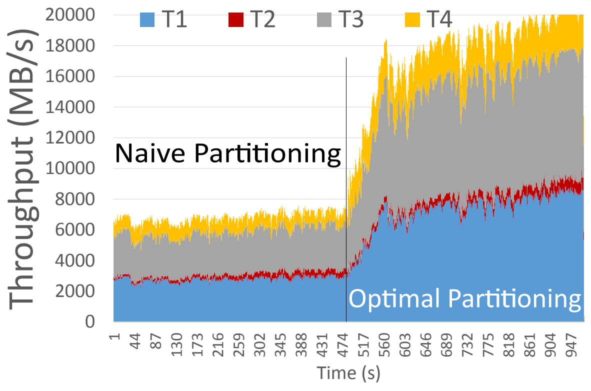
• **Lack of coordination.** Each cache in the IO stack makes its decisions locally, agnostic to the state of other caches in the stack, leading to inefficiencies, such as double caching, as was also noted by Wong and Wilkes [159].

• **Lack of adaptability.** Currently, the organization and configuration of caches is fixed. Caches cannot be added, removed, or resized on the fly to adapt to changes in the workload or in provider objectives.

• **Waste of system resources.** Simple solutions for partitioning caches along the IO stack are not sufficient. For example, Figure 5.2b shows that the observed performance triples when cache space is optimally allocated according to workload characteristics (the workload consists of 4 tenants using 120 VMs in total), compared to the case when caches are naively allocated across tenants. We will describe the details of this experiment in Section 5.5, but note that all workloads' throughputs benefit when the right cache size is chosen. This is true even for tenants that receive less total cache, as the contention at the storage device is reduced.



(a) The effect of eviction policy



(b) The effect of cache size

Figure 5.2: Performance depends on the cache policy (a) and allocation (b).

While some of these problems have been tackled in isolation, there is no comprehensive framework for the end-to-end management of caches that allows providers to address the major issues they are facing today. We present Moirai[1], a tenant- and workload-aware system that allows data center providers to

[1]Moirai (Ancient Greek for "apportioner") in Greek mythology are the three personifications of fate, who control the

control their distributed caching infrastructure to achieve provider objectives, such as improving resource utilization and request latency, achieving tenant isolation and QoS guarantees. Moirai does not require changes to the IO stack architecture, is transparent to applications and VMs, and does not change cache consistency semantics.

5.2 Design

We do not change the IO stack architecture, so IOs continue to flow along the same path as before. As a result, we do not pool or group caches together to provide cooperative caching or any new shared memory abstractions. Furthermore, it is our goal that our mechanisms do not change cache consistency semantics.

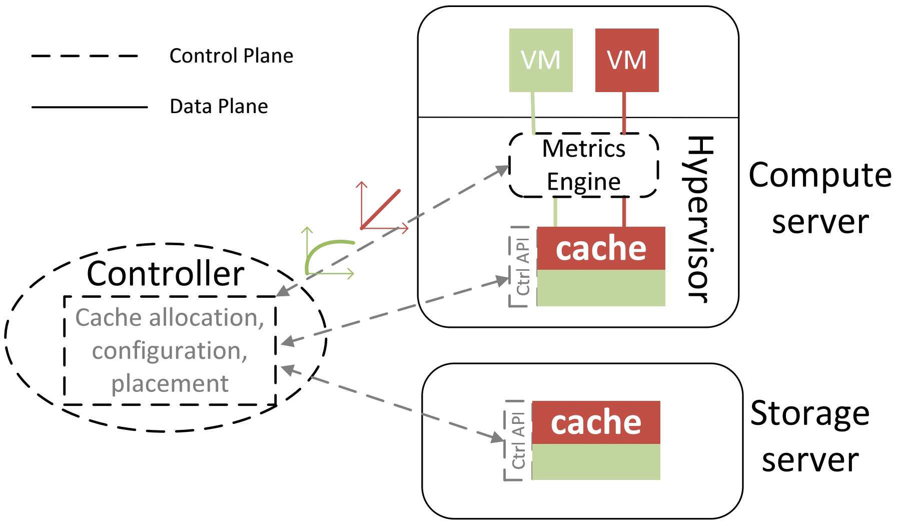


Figure 5.3: The Moirai architecture.

Figure 5.3 shows the architecture of Moirai, which comprises three key components:

1. The *Metrics Engine*, a hypervisor-based module that captures key characteristics for each workload.
2. Tenant-aware *programmable storage caches.*
3. A logically-centralized *controller* that uses information on workload characteristics to determine how much cache to allocate to each workload, where to place it, and what policies the cache should be configured with, then effects those changes on the data plane.

Details on each of the three components are provided next.

5.2.1 The Metrics Engine

The Metrics Engine is a hypervisor-based module that maintains key characteristics for each workload running on the system, such as throughput, number of reads vs. writes, and also *hit ratio curves*, which describe the percentage of requests serviced from cache as a function of the cache size. We use *phantom*

thread of life of every mortal from birth to death, analogously to the end-to-end control of caches by the three components that comprise Moirai.

`createCache` (<size,eviction pol,write pol>)
returns a reference to the newly created cache c
`removeCache` (Cache c)
`createRule` (IO Header h, Cache c)
creates cache rule <src,op,file,range>$\rightarrow c$
`removeRule` (IO Header h, Cache c)
`configureCache` (<size,eviction pol,write pol>, Cache c)
`getCacheStats` (Cache c)
returns cache statistics

Table 5.1: Moirai's API for a configurable cache.

caches, which inspect IO headers (with fields such as accessed file name, offset, length, etc.) and exploit techniques from recent work [158, 156, 123] to generate hit ratio curves efficiently at runtime. The Metrics Engine periodically sends these performance metrics to the centralized controller.

5.2.2 Programmable Caches

Caches along the IO stack are programmable through a simple API shown in Table 5.1. Caches are created at the desired position in the IO stack by sending a `createCache` call to the appropriate level in the stack (more details in Section 5.4). The `createRule` call is used to make a cache c workload-aware by specifying which IOs should be cached in c through the creation of rules to match incoming IO headers to the specified IO header entries. If an incoming IO matches its header with a rule, the IO (header + data) is sent through the cache. If no match is obtained, the IO bypasses the cache. The controller can also configure cache properties (`configureCache`) to set the size, eviction, and write policies. Similarly, cache performance metrics are obtained via the `getCacheStats` call.

Care must be taken to maintain consistency semantics when the location of a cache changes. For example, the controller could decide to cache at the storage server rather than at the hypervisor. In order to maintain consistency, Moirai first removes the caches on the old path, which automatically triggers the eviction of all cached state, including writing any dirty blocks to the back-end storage, and then installs caches on the new path.

5.2.3 Controller

The centralized controller uses the API described in Section 5.2.2 and information provided by the Metrics Engine to create and configure caches in order to implement a set of objectives specified by the provider, as illustrated in the next section.

5.3 Data Plane Transformations

In this section, we explore Moirai's ability to program and transform the data plane to implement various cloud provider objectives and improve workload performance. For each goal, we illustrate how the controller effects the necessary changes on the data plane.

5.3.1 Prioritizing a Workload

It's often desirable to be able to isolate the performance of a particular (high-priority) application A from that of another application B sharing a cache in the same VM. The controller can achieve this by configuring a dedicated cache C (a 50 GB LRU write-through cache in this particular example) inside the hypervisor, which is exclusive to workload A:

`C = createCache` ($<50GB$, LRU, write-through>)
`createRule` ($< VM$, *, A.file, *>, `C`)

The `createRule` call configures the cache to accept all R/W IOs originating from the VM, that access any part of A.file. The figure below shows the resulting data plane. Workload A flows through its own cache C in the hypervisor, while workload B continues along its previous path, bypassing that cache, effectively isolating A's traffic from it.

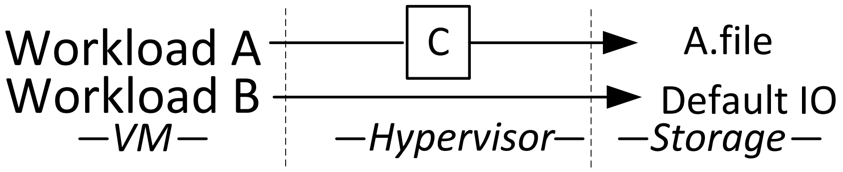


Figure 5.4: Isolating and prioritizing a workload's cache.

5.3.2 Providing Per-Workload Bandwidth Guarantees

Next we extend the objectives beyond simple priorities, and examine how Moirai allocates cache space to several arbitrary workloads $W_1, W_2, \ldots, W_n$, all running on the the same system, in order to guarantee each workload W_i a particular bandwidth B_i. Similar to Section 5.3.1, the controller passes each workload's traffic through its own dedicated cache C_i at the hypervisor (see Figure 5.5), but the question now becomes what the size each of the caches needs to be. In this section, we focus on hypervisor level caches only, but the techniques can be expanded to include simultaneous allocation of hypervisor and storage level cache space, as we explain in Section 5.3.5.

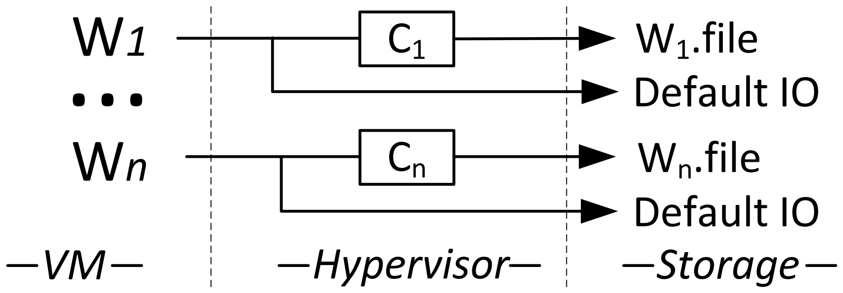


Figure 5.5: Allocating caches to multiple workloads.

To answer this question, the controller uses information from the Metrics Engine to first determine the hit ratio Hit_i^{cache} required for workload W_i to meet a certain bandwidth guarantee, and then allocates the workload W_i cache space a_i, such that $U(a_i) = Hit_i^{cache}$, where U is the workload's hit ratio function (provided by the Metrics Engine).

More precisely, note that if the total bandwidth achievable from the storage back-end [2] is $BW_i^{storage}$ and main memory bandwidth is BW^{memory}, a workload's bandwidth depends on its hit ratio Hit_i^{cache} as follows:

$$SLA_i^{BW} \leq Hit_i^{cache} \times BW^{memory} + (1 - Hit_i^{cache}) \times BW_i^{storage} \tag{5.1}$$

That means the cache hit ratio in order to achieve a bandwidth SLA_i^{BW} needs to be at least:

$$Hit_i^{cache} \geq \frac{SLA_i^{BW} - BW_i^{storage}}{BW^{memory} - BW_i^{storage}} \tag{5.2}$$

After the minimum data bandwidth guarantees $SLA_1^{BW}, \ldots, SLA_n^{BW}$ are met for all n workloads, the leftover cache space can be allocated based on priorities or using approaches highlighted in Section 5.3.3 to optimize for global utility.

5.3.3 Maximizing Global Workload Utility

Rather than per-workload guarantees, a provider might strive to maximize the global workload utility, i.e., the sum of the utilities across all workloads in the system. Utility of a workload could be measured by hit ratio, or be defined more generally in terms of bytes per second (Bps) satisfied by the cache, or by extending the notion of hit ratio by introducing weights to account for the type of IO (i.e. reads vs. writes), or even to account for the impact of a workload on the storage device (e.g. sequential vs. random access). The choice of definition for utility will be dictated by the optimization goals of the cloud provider.

Using the example of hit ratios as the utility function, the controller can create a separate cache for each workload (similar to Section 5.3.2) and then use a classic result [138] to determine the cache allocations $a_1, ..., a_n$. The algorithm, shown in Algorithm 5.3.1, uses a water-filling approach, i.e, it allocates the cache to workloads in small increments. The basic idea at each step is to allocate the next increment of cache to the workload that will achieve the highest hit rate out of the allocation. When the hit rate curves of workloads are *concave* functions, this algorithm will achieve an allocation that maximizes the total hit rate, i.e., total hit rate at the cache. We are currently investigating meta-heuristics to deal with non-concave hit ratio curves. For example, Soundararajan [134] proposed hill-climbing search, although we find that their particular algorithm and implementation is too slow for our system to react dynamically.

One might argue that a standard, workload-agnostic system that manages the entire cache as a single pool and applies its favourite replacement policy to it is also designed to achieve the same goal of maximizing overall hit ratio. However, Moirai can provide generalizations of this goal (e.g. a weighted sum of the hit ratios across workloads) and simultaneously provide other goals, such as isolation (e.g. protecting one workload from the effects of workload spikes in another workload), which a standard system cannot.

[2] If the storage back-end is remote, $BW_i^{storage}$ is the minimum of the network, and the back-end storage array's bandwidth.

```
Algorithm 5.3.1 Utility-maximizing cache allocation
Require: $n$ workloads sharing a cache of capacity $C$; $U_1$,...,$U_n$: hit rate curves for workloads.
Ensure: Assign cache allocation $a_i$ to workload $i$ s.t. $\sum a_i = C$, and max $\sum U(a_i)$
1: $\forall i, a_i = 0$  //Initialize allocations
2: $leftC = C$  //Cache left to distribute
3: $\varepsilon = 0.001 \times\ C$ //Water-filling constant (as fraction of C)
4: while ($leftC > 0$) do
5:     $cacheAlloc = \min(\varepsilon, leftC)$
6:     $j = \arg\max_i (U_i(a_i + cacheAlloc) - U_i(a_i))$ //workload with the most utility gained from extra cache
7:     $a_j += cacheAlloc$
8:     $leftC -= cacheAlloc$
```

5.3.4 Consolidating Memory Over Fast Networks

As systems are increasingly making use of fast networks with speeds in excess of 40-100 Gbps, and RDMA capabilities [38], use of remote resources is becoming increasingly feasible and can improve overall utilization of resources. Consider as an example a read-only dataset DATA.file accessed by N VMs across N hypervisors. Placing one consolidated cache at the storage server can result in an Nx reduction in total cache space used, with potentially only small increases in latency. The controller can accomplish this as follows (using the example of a 100 GB MRU write-back cache C as the consolidated cache):

```
C = createCache (<100 GB, MRU, write-back>)
createRule (< VM1 − N, *, DATA.file, *>, C)
```

The resulting data plane is shown in the figure below:

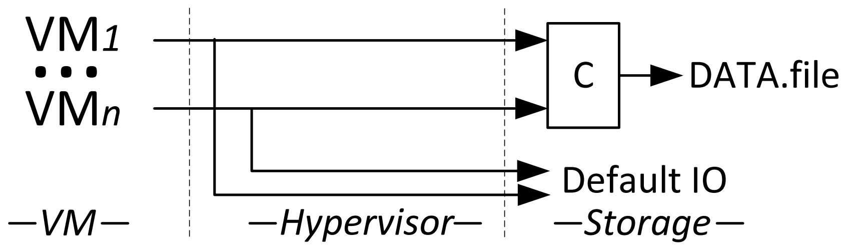


Figure 5.6: Consolidating memory at the storage server

5.3.5 Scaling Out Caches

In addition to fully-remote caching, caching capacity per workload can be split across the compute and storage server, while appearing to the VM and applications as one single aggregate cache. Note that today, workloads do flow through both caches (at the hypervisor, and at the storage server), but this occurs in an uncontrolled fashion, leading to wasted memory capacity by double-caching of data in both places.

In situations where the hypervisor is hosting several applications and memory is limited, the controller has several choices for how to split the cache for a workload A and configure it at the hypervisor(1) and storage server(2). If the workload access is uniform across the file, one choice is to cache half the file in each of the respective caches:

`createRule` (< VM, *, A.file, 0, size/2>, C1)
`createRule` (< VM, *, A.file, size/2+1, size>, C2)

The resulting data plane is shown in Figure 5.7:

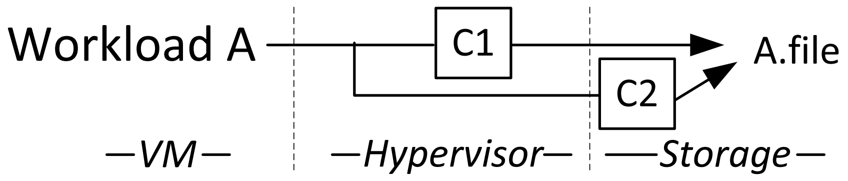


Figure 5.7: Scaling out a local cache

The controller can also match workload access patterns to the way the cache is split based on hot or cold blocks or files.

Another option is to treat both caches as a global LRU cache. To do that, the controller programs C1 to cache the IOs from A.file, and C2 to only cache IOs that were evicted (or "demoted") from C1. To provide per-workload bandwidth guarantees, Moirai extends the cache allocation method presented in Section 5.3.2. The controller now needs to determine two things:

1. How much cache space a_i to allocate for the global LRU cache made up of both C1 and C2, such that:
$$U(a_i) = Hit_i^{cache} \tag{5.3}$$
2. The individual cache space allocations a_i^1 and a_i^2, for C1 and C2 respectively. Thus, for some α:
$$\begin{aligned} a_i^1 &= \alpha \times a_i \\ a_i^2 &= (1-\alpha) \times a_i \end{aligned} \tag{5.4}$$

The relationship between these variables is illustrated using a simple, example hit rate utility function in Figure 5.8.

A workload's bandwidth SLA_i^{BW} now depends on the hit ratio Hit_i^{cache} of the global LRU cache as follows:

$$SLA_i^{BW} \leq Hit_i^{cache} \times BW_i^{GlobalLRU} + (1 - Hit_i^{cache}) \times BW_i^{storage} \tag{5.5}$$

Similar to Equation (5.2), the cache hit ratio Hit_i^{cache} needs to be at least:

$$Hit_i^{cache} \geq \frac{SLA_i^{BW} - BW_i^{storage}}{BW_i^{GlobalLRU} - BW_i^{storage}} \tag{5.6}$$

Here, $BW_i^{GlobalLRU}$ refers to the total bandwidth achievable from the global LRU cache.

Since C1 and C2 form the global LRU cache, the fraction α of cache space allocated to C1 will result in $U(\alpha a_i)$% of the hits, while the rest of the hits, $[U(a_i) - U(\alpha a_i)]$%, will be served from C2. Since C1 is a hypervisor cache, its achievable bandwidth is BW^{memory}, while C2's achievable bandwidth is constrained by the bandwidth of the network $BW_i^{network}$. Thus:

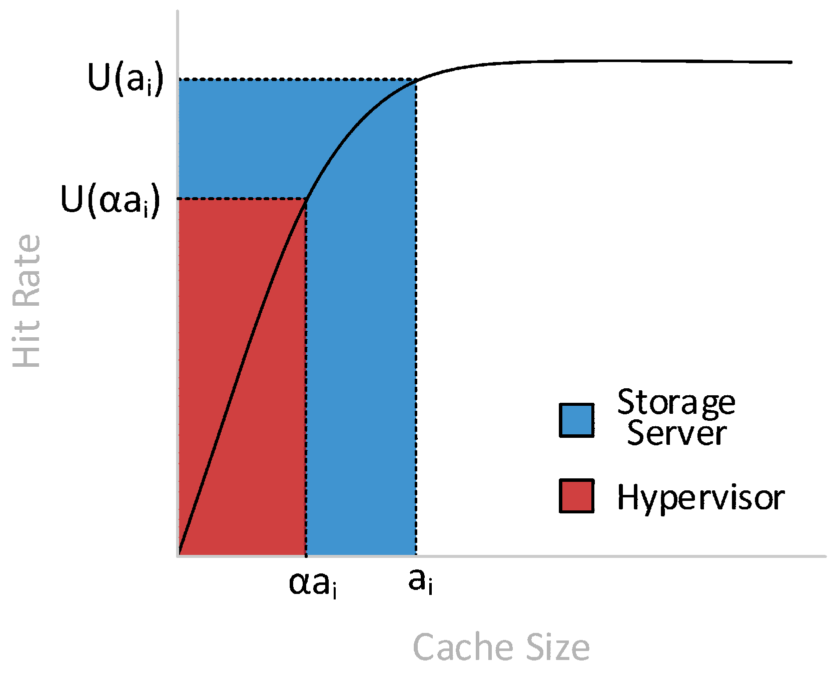


Figure 5.8: Example of a cache hit rate function U, and associated parameters a_i and α, used to compute cache allocations for scaled-out LRU caches with bandwidth guarantees.

$$\begin{aligned} BW_i^{GlobalLRU} &= U(\alpha a_i) \times BW^{memory} \\ &+ [U(a_i) - U(\alpha a_i)] \times BW_i^{network} \end{aligned} \tag{5.7}$$

Simultaneously using Equations (5.3), (5.4), (5.6), and (5.7), the controller solves for a_i, and α, effectively determining the cache allocations a_i^1 and a_i^2, for C1 and C2 respectively. Further constraints can also be added to the problem statement (e.g., imposing a maximum size on either C1, or C2) to limit the solution space.

5.4 Implementation

We have implemented and deployed a Moirai prototype, comprising all components described in Section 5.2, on a Windows-based system and made the code publicly available [92]. The controller is implemented in around 6000 LOC of C# and communicates with the caches through RPCs over TCP. The Metrics Engine is implemented as a user-level stage in the hypervisor in around 500 LOC and uses a variant of SHARDS [156] to determine hit ratio curves. Cache modules implement the APIs in Table 5.1 at user-level in around 2000 LOC in C#.

One implementation challenge is how to classify and direct a tenant's traffic to the configurable caches. We decided to build an extension of the IOFLow framework [147] to implement this functionality. Note that while in its original form IOFlow does keep track of each IO's tenant class, it was designed to provide IO queueing and rate limiting based on IO request headers. By contrast, caching involves inspection and manipulation of the *data* associated with an IO request. We implemented an extension of the IOFlow architecture to add support for data transformations using a version of the Windows messaging API for

	H.Index	H.Data	H.Msg	H.Log	Exchange
Read %	75%	61%	56%	1%	40%
IO Sizes	64 KB	8 KB	64 KB	64 KB	8 KB
Seq/rand	Mixed	Rand	Rand	Seq	Rand
# IOs	32M	158M	36M	54M	60M

Table 5.2: Characteristics for 4 Hotmail workloads, part of a 2-day Hotmail IO trace and an Exchange workload, part of a 1-day Exchange IO trace. Seq/rand refer to sequential and random-access respectively. M=million.

filter drivers, in around 500 new LOC. IOs are passed to a user-level cache through an upcall, while a kernel-mode thread handling the I/O request blocks pending a return code from the cache. The latter decides whether the request is terminated at the filter driver (hit), or is sent further down the IO stack.

5.5 Experimental Evaluation

This section provides an experimental evaluation of some of Moirai's use cases presented in Section 5.3. Our experimental testbed has 12 servers, each with 16 Intel Xeon 2.4 GHz, 384 GB of RAM and three Seagate Constellation 2 disks or four Intel 520 SSDs in RAID-0. The servers run Windows Server 2012 R2 operating system and can act as either Hyper-V hypervisors or as storage servers. Each server has a 40 Gbps Mellanox ConnectX-3 NIC supporting RDMA and connected to a Mellanox MSX1036B-1SFR switch. We use a combination of real enterprise application traces and benchmarks, as specified in more detail below. As Moirai is transparent to applications and VM's, they can run on our testbed without modifications

We use a mixture of real enterprise application traces and benchmarks in the evaluation. For the former, we use public traces from an enterprise Exchange email server [133] and Hotmail [148]. Key characteristics of these traces are shown in Table 5.2. The traces are diverse across a number of metrics such as the Read-to-Write ratio, IO sizes, sequentiality of access and number of IOs which allows for a comprehensive evaluation across realistic workload mixes. However, a limitation of these workloads is that they were originally collected *underneath* file caches. As such, they under-represent the amount of application reads.

To account for this limitation, we also use TPC-E [152] and TPC-H [151] to cover a broad class of workloads, ranging from transaction processing OLTP operations with small IO sizes (TPC-E) to large streaming IO from data mining queries (TPC-H). They run over unmodified SQL Server 2012 R2 databases. When error bars are shown they represent the average, minimum and maximum from 5 runs.

5.5.1 Enforcing Priorities

We examine Moirai's ability to *prioritize* a workload using the example of a VM with one SQL Server instance running both TPC-E and TPC-H. The corresponding database files, "TPCE.VHD" and "TPCH.VHD" each have a footprint of 50 GB and are stored on *Virtual Hard Drives* (VHDs) on two separate disk-based storage servers.

We run two experiments, one with default caching and one where we use Moirai to prioritize the TPC-E workload, as explained in Section 5.3.1 and measure the throughput (transactions/min) for the TPC-E workload. The results are shown in Figure 5.9.

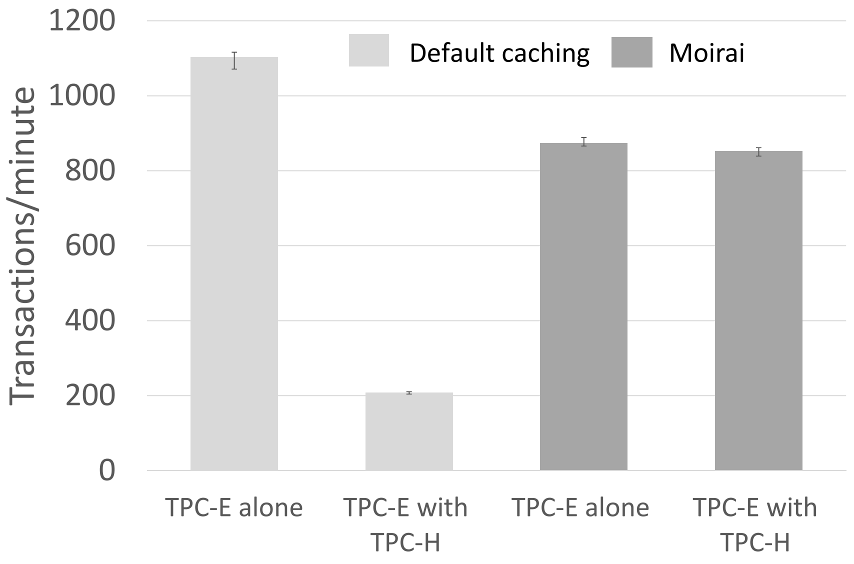


Figure 5.9: Prioritizing one workload (TPC-E) vs another (TPC-H). With Moirai, the performance of TPC-E is not impacted by TPC-H. In contrast, today, running both workloads together would result in a 5x performance hit for TPC-E.

We observe that in the system without Moirai, TPC-E's performance drops by more than 5X when TPC-H runs. On the other hand, we find that with Moirai, TPC-E's throughput running alongside TPC-H is within 2.3 % of its throughput running by itself.

Note that our current implementation of Moirai results in a data plane overhead of 20% for workload throughput (this difference is due to using our user-level cache vs. SQL Server's native cache, which is heavily optimized). The overhead stems in part from extra memory copies between the kernel and the user-level cache. However, we believe that this overhead is acceptable compared to the 5x drop in performance with today's caching infrastructure. Further, note that the controller can detect when no other workloads are running and remove the user-level cache and thus avoid the extra overhead.

5.5.2 Maximizing Global Hit Rate

We consider the example of maximizing global hit rate using four tenants with 30 VMs each, spread over 10 hypervisors accessing VHDs on an SSD-based storage server. Each tenant's VM uses IOMeter, parameterized with the key characteristics of the Hotmail workloads (Tenants 1-4 are running the Index, Data, Msg and Log workloads respectively).

The experiment has two phases. In the first phase, naive caching is used, where each tenant receives an equal amount of cache. Note that tenant 4 represents the log workload - any cache it consumes is wasted (0% hit rate) because the workload is sequential. In the second phase, Moirai is enabled, and it uses the method described in Section 5.3.3 to partition the cache and reconfigure the data plane. The results are shown in Figure 5.10.

Interestingly, we observe not only that overall throughput increases by more than 2.5x, but also that this improvement comes at no cost to any of the individual four tenants. The reason is that all tenants benefit from the decreased load at the storage back-end.

We have experimented with other workload combinations as well. In the worst case across all experiments the overall throughput still increased by 35%, but this came at the cost of a small penalty to one

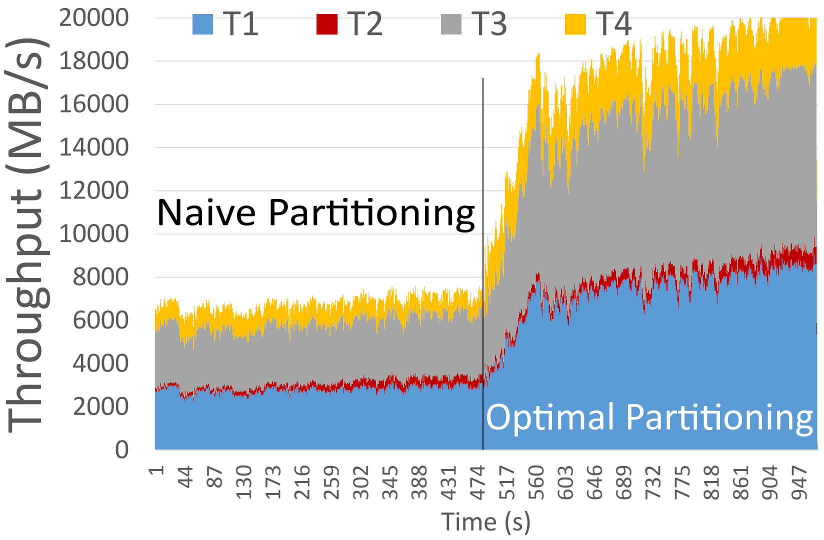


Figure 5.10: Maximizing global hit rate for 4 tenants.

tenant, whose throughput dropped by 10%. A cloud provider could feed into the controller a minimum amount of cache space or minimum hit rate it wants to guarantee each workload, and then ask it to divide the remaining cache space to maximize global utility.

5.5.3 Consolidating Memory Over Fast Networks

In this section we use Moirai on a TPC-H workload running on ten different hypervisors to illustrate the trade-offs for memory consolidation over fast networks. We compare the case where Moirai is used to insert a 50 GB cache inside each of the 10 hypervisors, to the case where Moirai inserts one shared 50 GB cache at the storage server, which is either accessed at 1Gbps over TCP or at 40Gbps over RDMA. In all cases, all the data resides in memory (100% hit rate). The results are shown in Figure 5.11.

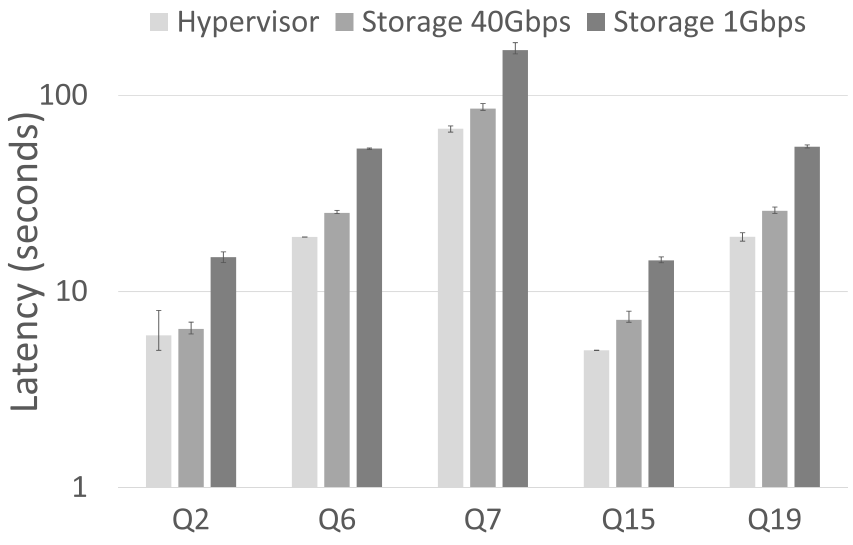


Figure 5.11: Latency for 5 TPC-H queries. The controller can decide to use file caches in the storage server for fast RDMA-based networks. Y-axis is log scale.

We observe that the average latency overhead when using a consolidated cache over the fast network is around 26%, compared to using local hypervisor caches. For the slow network the overheads are 153%. Note that in exchange for paying these overheads one gains a 10X reduction in the total amount of cache space allocated for this workload. Such savings can be passed to tenants for consuming less resources as well. Also note that with Moirai a provider has the option to seamlessly switch from one cache configuration to another, depending on the state of the system. For example, a provider might switch to a consolidated cache at the cost of some latency penalties when cache space is scarce.

5.5.4 Scaling Out Caches

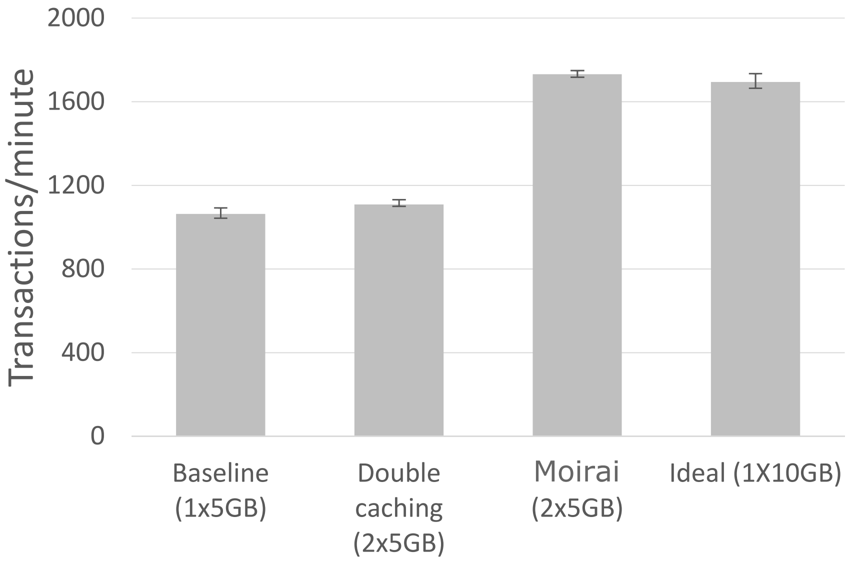


Figure 5.12: Splitting IOs for TPC-E across two different caches. Today, "double caching" occurs since all IOs flow through all caches. Moirai can prevent this, and match the performance of an aggregate cache.

In this section, we evaluate Moirai's ability to scale out the storage cache as described in Section 5.3.5. We consider a TPC-E workload on a machine low on memory. The provider wishes to scale out TPC-E's 5 GB hypervisor cache to include another 5 GB at the storage server.

Figure 5.12 shows the performance of TPC-E in the experiment. The first bar shows TPC-E's baseline performance with 5 GB of hypervisor-only cache. The second bar shows another baseline today, where 5 GB of cache are used in both compute and storage server, but because all IOs take the same path, the second cache is ineffective due to double caching. The third bar shows that TPC-E performance increases by 63% over the baseline hypervisor-only cache when Moirai is in use. Finally, the last bar shows that using Moirai is as efficient as the case when the hypervisor cache is given 10 GB of memory.

5.5.5 Dynamic Workloads

The controller in Moirai continuously monitors the metrics and dynamically reacts to changes after some reaction time s, a configurable parameter. For example, Moirai will detect when a cache goes unutilized and reuse the space accordingly. We have worked with values for s on the order of 15-30 seconds - we believe that this range presents a good trade-off between responsiveness and unwarranted reconfigurations due to momentary changes in workload demand.

To illustrate Moirai's dynamic capabilities, in Figure 5.13, we evaluate a setup consisting of 10 hypervisors each with 12 VMs, where each VM has a 2GB file stored on an SSD back-end that it accesses through IOMeter.

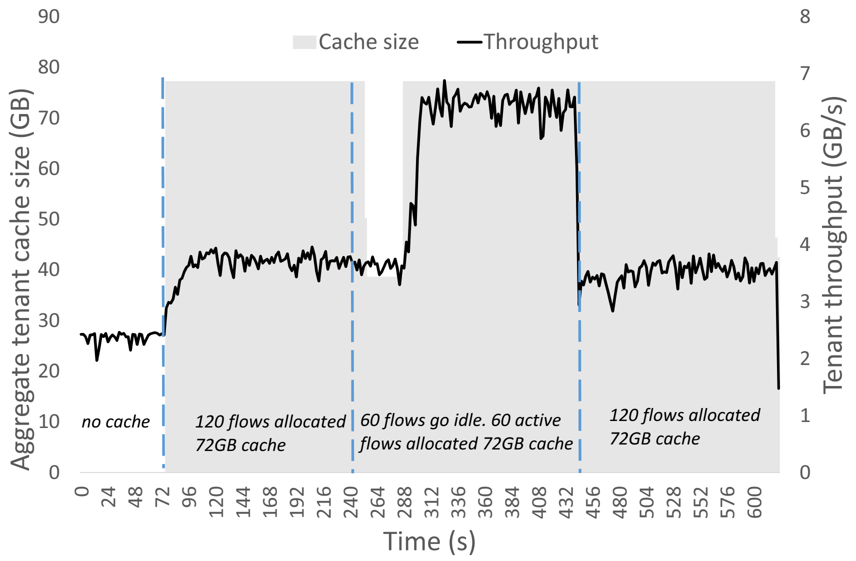


Figure 5.13: Moirai adapting to workloads dynamically over time. Note there are two y-axis.

In the first phase, the VMs run with no cache. In the second phase, the controller allocates a total of 72GB cache at the hypervisor level, which is evenly split between VMs (i.e., each receiving 72/120 GB), and passes the 120 VMs' traffic through 10 user-level caches (one in each hypervisor machine). The application's throughput improves because the cache hit rate increases to 30%. In the third phase, half of the VMs finish their work and go idle. The aggregate used cache size drops. The controller detects that drop and increases the size of the remaining 60 VM caches. In the hypervisors with the inactive VMs, the unused cache space is released to the OS and can be used by other applications. The total throughput increases as the 60 VMs get a 60% hit rate. In the final phase, all VMs again become active, triggering a re-configuration of the cache sizes by the controller. The decreased cache size at each hypervisor results in a lower overall hit rate as expected.

5.5.6 Control Plane Overheads

We now consider Moirai's overheads on the control plane. Moirai implements a version of SHARDS [156] in the Metrics Engine to construct hit ratio curves at runtime. Our current implementation uses up to 100% of one core. While our current implementation is not as highly-optimized, the original SHARDS paper showed that hit ratio curves with very high fidelity can be constructed online using less than 10MB of memory per workload [156], and marginal (less than 5%) CPU overhead. We believe these overheads are very reasonable.

We also evaluated the time it takes to compute the optimal cache size allocation as a function of the number of VMs in the system (Algorithm 3.1, described in Section 3.3). We varied the number of VMs from 100 to 5000, and measured the time it took to compute the allocation. For 5000 VMs, it took less than 15s to make that decision, with a water-filling constant ϵ of 1MB. For ϵ of 2MB, the time is less than 5s. This highlights the tradeoff between how fine-grained the cache allocation is, and the completion time for the allocation algorithm. However, since cache allocations can feasibly be done at granularities (ϵ) coarser than 2MB, and they do not need to be re-computed at very short time intervals, we believe this method of cache allocation is very reasonable. We are currently exploring optimizations to reduce the runtime further.

5.6 Summary

Caches are a critical resource in data centers. They improve latency, throughput and reduce the load on networks and storage. But today, caches are implicit, not designed for controlled sharing, leading to severe inefficiencies under multi-tenancy. This chapter presents Moirai, a software-defined caching architecture that enables control of caches in a multi-tenant data center. Moirai is transparent to hosted tenants. Their throughput and latency benefit without requiring any tenant input or hints. We show using several different use cases how Moirai can help ease the management of the distributed caching infrastructure and enable the provider to achieve a series of different objectives. We hope that our public release of the code [92] implementing Moirai will help foster future work in this area.

Chapter 6

Treating the Storage Stack Like a Network

6.1 Introduction

An application's IO stack is rich in stages providing compute, network, and storage functionality. These stages include guest OSes, file systems, hypervisors, network appliances, and distributed storage with caches and schedulers. There are over 18+ types of stages on a typical data center IO stack [147]. Furthermore, most IO stacks support the injection of new stages with new functionality using filter drivers common in most OSes [95, 46, 84], or appliances over the network [129].

Controlling or programming how IOs flow through this stack is hard if not impossible, for tenants and service providers alike. Once an IO enters the system, the path to its endpoint is pre-determined and static. It must pass through all stages on the way to the endpoint. A new stage with new functionality means a longer path with added latency for every IO. As raw storage and networking speeds improve, the length of the IO stack is increasingly becoming a new bottleneck [111]. Furthermore, the IO stack stages have narrow interfaces and operate in isolation. Unlocking functionality often requires coordinating the functionality of multiple such stages. These reasons lead to applications running on a general-purpose IO stack that cannot be tuned to any of their specific needs, or to one-off customized implementations that require application and system rewrite.

This chapter's main contribution is experimenting with applying a well-known networking primitive, *routing*, to the storage stack. IO routing provides the ability to dynamically change the path and destination of an IO, like a `read` or `write`, at runtime. Control plane applications use IO routing to provide customized data plane functionality for tenants and data center services.

Consider three specific examples of how routing is useful. In one example, a load balancing service selectively routes `write` requests to go to less-loaded servers, while ensuring `read` requests are always routed to the latest version of the data (Section 6.5.1). In another example, a control application provides per-tenant throughput versus latency tradeoffs for replication update propagation, by using IO routing to set a tenant's IO read- and write-set at runtime (Section 6.5.2). In a third example, a control application can route requests to per-tenant caches to maintain cache isolation (Section 6.5.3).

IO routing is challenging because the storage stack is stateful. Routing a `write` IO through one path to endpoint A and a subsequent `read` IO through a different path or to a different endpoint B needs to be mindful of application consistency needs. Another key challenge is data plane efficiency. Changing the path of an IO at runtime requires picking a point on the data plane the path is changed in order to minimize the number of hops an IO takes, as well as to minimize IO processing times.

sRoute's approach builds on the IOFlow storage architecture [147]. IOFlow already provides a separate control plane for storage traffic and a logically centralized controller with global visibility over the data center topology. As an analogy to networking, sRoute builds on IOFlow just like software-defined networking (SDN) functions build on OpenFlow [87]. IOFlow also made a case for request routing. However, it only explored the concept of *bypassing* stages along the IO path, and did not consider the full IO routing spectrum where the path and endpoint can also change, leading to consistency concerns. This chapter provides a complete routing abstraction.

This chapter makes the following contributions:

- We propose an IO routing abstraction for the IO stack.
- sRoute provides per-IO and per-flow routing configuration updates with strong semantic guarantees.
- sRoute provides an efficient control plane. It does so by distributing the control plane logic required for IO routing using *delegate functions*.
- We report on our experience in building three control applications using IO routing: tail latency control, replica set control, and file caching control.

The results of our evaluation demonstrate that data center tenants benefit significantly from IO stack customization. The benefits can be provided to today's unmodified tenant applications and VMs. Furthermore, writing specialized control applications is straightforward because they use a common IO routing abstraction.

6.2 Background

The data plane, or *IO stack* comprises all the stages an IO request traverses from an application until it reaches its destination. For example, a read to a file will traverse a guest OS' file system, buffer cache, scheduler, then similar stages in the hypervisor, followed by OSes, file systems, caches and device drivers on remote storage servers. We define per-IO routing in this context as the ability to control the IO's endpoint as well as the path to that endpoint. The first question is what the above definition means for storage semantics. A second question is whether IO routing is a useful abstraction.

To address the first question, we looked at a large set of storage system functionalities and distilled from them three types of IO routing that make sense semantically in the storage stack. Figure 6.1 illustrates these three types. In *endpoint* routing, IO from a source p to a destination file X is routed to another destination file Y. In *waypoint* routing, IOs from sources p and r to a file X are first routed to a specialized stage W. In *scatter* routing, IOs from p and r are routed to a subset of data replicas.

This chapter makes the case that IO routing is a useful abstraction. We show that many specialized functions on the storage stack can be recast as routing problems. Our hypothesis when we started this work was that, because routing is inherently programmable and dynamic, we could substitute hard-coded one-off implementations with one common routing core. Table 6.1 shows a diverse set of such storage stack functionalities, categorized according to the type of IO routing that enables them.

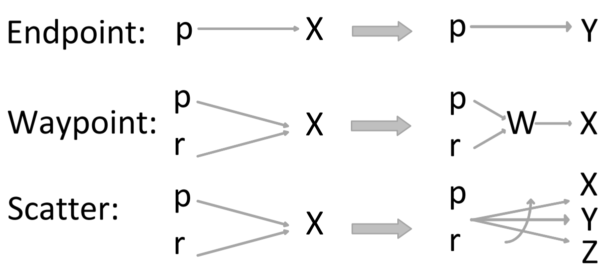


Figure 6.1: Three types of IO routing: endpoint, waypoint and scatter. p, r refer to sources such as VMs or containers. X, Y, Z are endpoints such as files. W represents a waypoint stage with specialized functionality, for example a file cache or scheduler.

	Functionality	How IO routing helps
Endpoint	Tail latency control	Route IO to less loaded servers
	Copy-on-write	Route writes to new location
	File versioning	Route IO to right version
Waypoint	Cache size guarantee	Route IO to specialized cache
	Deadline policies	Route IO to specialized scheduler
Scatter	Maximize throughput	Route `reads` to all replicas
	Minimize latency	Route `writes` to replica subset
	Logging/Debugging	Route selected IOs to loggers

Table 6.1: Examples of specialized functionality and the type of IO routing that enables them.

Endpoint routing. Routes IO from a multi-source application $\{p, r\}$ to a file X to another file Y. The timing of the routing and operation semantics is dictated by the control logic. For example, write requests could go to the new endpoint and reads could be controlled to go to the old or new endpoints. Endpoint routing enables functionality such as improving *tail latency* [33, 101], *copy-on-write* [107, 57, 122], *file versioning* [93], and *data re-encoding* [2]. These policies have in common the need for a dynamic mechanism that changes the endpoint of new data and routes IO to the appropriate endpoint. Section 6.5.1 shows how we implement tail latency control using endpoint routing.

Waypoint routing. Routes IO from a multi-source application $\{p, r\}$ to a file X through an intermediate waypoint stage W. W could be a file cache or scheduler. Waypoint routing enables specialized appliance processing [129]. These policies need a dynamic mechanism to inject specialized waypoint stages or appliances along the stack and to selectively route IO to those stages. Section 6.5.3 shows how we implement file cache control using waypoint routing.

Scatter routing. Scatters IO from file X to additional endpoints Y and Z. The control logic dictates which subset of endpoints to read data from and write data to. Scatter routing enables specialized *replication* and *erasure coding* policies [145, 78], as well as interactive logging and debugging of the storage stack[9, 132, 45]. These policies have in common the need for a dynamic mechanism to choose which endpoint to write to and read from. This control enables programmable tradeoffs around throughput and update propagation latency. Section 6.5.2 shows how we implement replica set control using scatter routing.

6.2.1 Challenges

IO routing is challenging for several reasons:

Consistent system-wide configuration updates. IO routing requires a control-plane mechanism for changing the path of an IO request. The mechanism needs to coordinate the forwarding rules inserted into the data plane to change the path of IOs. Any configuration changes must not lead to system instability, where an IO's semantic guarantees are violated by having it flow through an incorrect path.

Metadata consistency. IO routing allows `read` and `write` IOs to be sent to potentially different endpoints. Several applications benefit from this flexibility. Some applications, however, have stricter consistency requirements and require, for example, that a `read` always follow the path of a previous `write`. A challenge is keeping track of the data's latest location. Furthermore, IO routing metadata needs to coexist consistently with metadata in the rest of the system. The guest file system, for example, has a mapping of files to blocks and the hypervisor has a mapping of blocks to virtual disks on an (often) remote storage backend. The backend could be a distributed system of its own with a metadata service mapping files or chunks to file systems to physical drives.

File system semantics. Some file system functionality (such as byte-range file locking when multiple clients access the same file) depends on consulting file system state to determine the success and semantics of individual IO operations. The logic and state that dictates the semantics of these operations resides inside the file system, at the destination endpoint of these IOs. IO routing needs to maintain the same file system functionality and semantics in the storage stack.

Efficiency. Providing IO stack customization requires a different way of building specialized functionality. We move away from an architecture that hard-codes functionality on the IO stack to an architecture that dynamically directs IOs to specialized stages. Any performance overheads incurred must be minimal.

6.3 Design

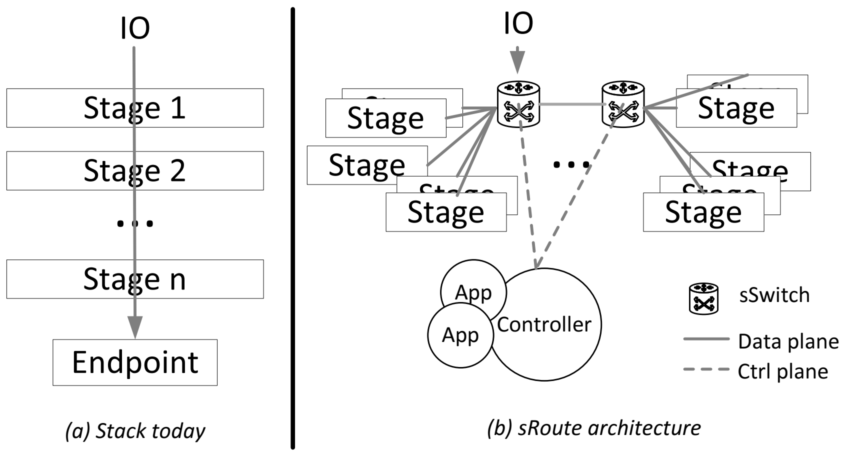


Figure 6.2: System architecture. sSwitches can route IO within a physical machine's IO stack and across machines over the network.

Figure 6.2 shows sRoute's architecture. It is composed of **sSwitches** on the data plane, that change the route of IOs according to forwarding rules. sSwitches are programmable through a simple API with four calls shown in Table 6.2. The sSwitches forward IOs to other file destinations, the controller, or to specialized stages (e.g., one that implements a particular caching algorithm). **A control plane** with a logically-centralized controller specifies the location of the sSwitches and inserts forwarding rules in them. Specialized **stages** take an IO as an input, perform operations on its payload and return the IO back to the sSwitch for further forwarding.

6.3.1 Baseline architecture

The baseline system architecture our design builds on is that of an enterprise data center. Each tenant is allocated VMs or containers[1] and runs arbitrary applications or services in them. Network and storage are virtualized and VMs are unaware of their topology and properties.

The baseline system is assumed to already have separate control and data planes and builds on the IOFlow architecture [147]. That architecture provides support for flow-based classification and queuing and communication of basic per-flow statistics to a controller.

6.3.2 Design goals

sRoute's design targets several goals. First, we want a solution that does not involve application or VM changes. Applications have limited visibility of the data center's IO stack. This chapter takes the view that data center services are better positioned for IO stack customization. These are then exposed to applications through new types of service level agreements (SLA), e.g., guaranteeing better throughput and latency. Second, data-plane performance overheads should be minimal. Third, the control plane should be flexible and allow for a diverse set of application policies.

The rest of this section focuses on the sSwitches and the control plane interfaces to them. Section 6.4 presents implementation details. Section 6.5 focuses on control applications. Figure 6.3 provides the construct definitions used in the rest of the chapter.

6.3.3 sSwitches on the data plane

An sSwitch is a special stage that is inserted into the IO stack (data plane) to provide IO routing. An sSwitch forwards IO according to rules specified by the control plane. A forwarding rule contains two parts: an IO header and an action or delegate function[2]. IO packets are matched against the IO header, and the associated delegate in the *first* successful rule match executes (hence, the order of installed rules matters). In the simplest form, this delegate returns a set of stages where the IO should next be directed. For example, routing all traffic from VM_1 for file X on server S_1 to file Y on server S_2 can be represented with this rule:

1: $< VM_1, *,\ //S_1/X > \rightarrow (; return\{< IO,\ //S_2/Y >\})$

An sSwitch implements four control plane API calls as shown in Table 6.2. The APIs allow the control plane to `Insert` a forwarding rule, or `Delete` it. Rules can be changed dynamically by two entities on the control plane: the controller, or a control `Delegate` function.

As defined in Figure 6.3, the IO header is a tuple containing the source of an IO, the operation, and the file affected. The source of an IO can be a process or a VM uniquely authenticated through

[1]This chapter's implementation uses VMs.
[2]The reason the second part of the rule is a function (as opposed to simply a set of routing locations) is for control plane efficiency in some situations, as is explained further in this section.

`Insert` (IOHeader, Delegate) Creates a new fwd. rule matching the IO header, using dynamic control delegate to look up destination
`Delete` (IOHeader) Deletes all rules matching the header
`Quiesce` (IOHeader, Boolean) Blocks or unblocks incoming IO matching IO header when Boolean is true or false respectively
`Drain` (IOHeader) Drains all pending IOs matching the IO header

Table 6.2: Control API to the sSwitch.

Rule	:=	$IOHeader \rightarrow Delegate(IOHeader)$
IOHeader	:=	$< Source, Operation, File >$
Delegate	:=	$F(IOHeader); return\{Detour\}$
Source	:=	Unique Security Identifier
Operation	:=	`read`|`write`|`create`|`delete`
File	:=	$< FileName, Offset, Length >$
Detour	:=	$< IO\|IOHeader, DetourLoc >$
DetourLoc	:=	$File\|Stage\|Controller$
Stage	:=	$< HostName, DriverName >$
F	:=	Restricted code

Figure 6.3: Construct definitions.

a security identifier. The destination is a file in a (possibly remote) share or directory. Building on IOFlow's classification mechanism [147] allows an sSwitch to have visibility over all this information and other relevant IO header entries at any point in the IO stack (without IOFlow, certain header entries such as the source, could be lost or overwritten as IO flows through the system).

The operation can be one of `read`, `write`, `create` or `delete`. Wildcards and longest prefix matching can be used to find a match on the IO header. A default match rule sends an IO to its original destination. A detour location could be a file (e.g., another file on a different server from the original IO's destination), a stage on the path to the endpoint (example rule 1 below), or the centralized controller (example rule 2 below that sends the IO header for all writes from VM_2 to the controller):

1: $< VM_1, *,\ //S_1/X > \rightarrow (; return\{< IO,\ //S_2/C >\})$
2: $< VM_2, w, * > \rightarrow (; return\{< IOHeader,\ Controller >\})$

The sSwitch is responsible for transmitting the full IO or just its header to a set of stages. The response does not have to flow through the same path as the request, as long as it reaches the initiating source[3].

Unlike in networking, the sSwitch needs to perform more work than just forwarding. It also needs to prepare the endpoint stages to accept IO, which is unique to storage. When a rule is first installed, the sSwitch needs to open a file connection to the endpoint stages, in anticipation of IO arriving. The sSwitch needs to create it and take care of any namespace conflicts with existing files. `Open` and `create` operations are expensive synchronous metadata operations. There is an inherent tradeoff between lazy

[3]sSwitches cannot direct IO responses to sources that did not initiate the IO. Finding scenarios that need such source routing and the mechanism for doing so is future work.

file creation upon the first IO arriving and file creation upon rule installation. The former avoids unnecessarily creating files for rules that do not have any IO matching them, but upon a match the first IO incurs a large latency. The latter avoids the latency but could create several empty files. The exact tradeoff penalties depend on the file systems used. By default this chapter implements the latter, but ideally this decision would also be programmable (but it is not so yet.)

sSwitches implement two additional control plane APIs. A `Quiesce` call is used to block any further requests with the same IO header from propagating further. The implementation of this call builds on the lower-level IOFlow API that sets the token rate on a queue [147]. `Drain` is called on open file handles to drain any pending IO requests downstream. Both calls are synchronous. These calls are needed to change the path of IOs in a consistent manner, as discussed in the next section.

6.3.4 Controller and control plane

A logically centralized controller has global visibility over the stage topology of the data center. This topology comprises all physical servers, network and storage components as well as the software stages within a server. Maintaining this topology in a fault-tolerant manner is already feasible today [63].

The controller is responsible for three tasks. First, it takes a high level tenant policy and translates it into sSwitch API calls. Second, it decides *where* to insert the sSwitches and specialized stages in the IO stack to implement the policy. Third, it disseminates the forwarding rules to the sSwitches. We show these tasks step-by-step for two simple control applications below.

The first control application directs a tenant's IO to a **specialized file cache**. This policy is part of a case study detailed in Section 6.5.3. The tenant is distributed over 10 VMs on 10 different hypervisors and accesses a read-only dataset X. The controller forwards IO from this set of VMs to a specialized cache C residing on a remote machine connected to the hypervisors through a fast RDMA network. The controller knows the topology of the data paths from each VM to C and injects sSwitches at each hypervisor. It then programs each sSwitch as follows:

```
1: for $i \leftarrow 1, 10$ do
2:     Quiesce ($< VM_i$, *, //$S_1$/X>, true)
3:     Drain ($< VM_i$, *, //$S_1$/X>)
4:     Insert ($< VM_i$, *, //$S_1$/X>, (; return {<IO, //$S_2$/C>}))
5:     Quiesce ($< VM_i$, *, //$S_1$/X>, false)
```

Lines 2 and 3 are needed to complete any IOs in-flight. This is done so that the sSwitch does not need to keep any extra metadata to know which IOs are on the old path. That metadata would be needed, for example, to route a newly arriving `read` request to the old path since a previous `write` request might have been buffered in an old cache on that path. The delegate on line 4 simply returns the cache stage. Finally, line 5 unblocks IO traffic. The controller also injects an sSwitch at server S_2 where the specialized cache resides, so that any requests that miss in cache are sent further to the file system of server S_1. The rule at S_2 matches IOs from C for file X and forwards them to server S_1:

```
1: Insert ($< C$, *, //$S_1$/X>, (; return {<IO, //$S_1$/X>}))
```

The second control application improves a tenant's **tail latency** and illustrates a more complex control delegate. The policy states that queue sizes across servers should be balanced. This policy is part of a case study detailed in Section 6.5.1. When a load burst arrives at a server S_1 from a source VM_1 the control application decides to temporarily forward that load to a less busy server S_2. The controller can choose to insert an sSwitch in the VM_1's hypervisor or at the storage server S_1. The latter means that IOs go to S_1 as before and S_1 forwards them to S_2. To avoid this extra network hop the controller chooses the former. It then calls the following functions to insert rules in the sSwitch:

```
1: Insert (<$VM_1$, w, //$S_1$/X>, (F(); return <IO, //$S_2$/X>))
2: Insert (<$VM_1$, r, //$S_1$/X>, (return <IO, //$S_1$/X>))
```

The rules specify that `writes` "w" are forwarded to the new server, whereas `reads` "r" are still forwarded to the old server. This application demands that `reads` return the latest version of the data. When subsequently a `write` for the first 512 KB of data arrives[4], the delegate function updates the read rule through function $F()$ whose body is shown below:

```
1: Delete (<$VM_1$, r, //$S_1$/X>)
2: Insert (<$VM_1$, r, //$S_1$/X, 0, 512$KB$ >, (return <IO, //$S_2$/X>))
3: Insert (<$VM_1$, r, //$S_1$/X>, (return <IO, //$S_1$/X>)
```

Note that quiescing and draining are not needed in this scenario since the sSwitch is keeping the metadata necessary (in the form of new rules) to route a request correctly. A subsequent `read` for a range between 0 and 512 KB will match the rule in line 2 and will be sent to S_2. Note that sSwitch matches on byte ranges as well, so a `read` for a range between 0 and 1024 KB will be now split into two reads. The sSwitch maintains enough buffer space to coalesce the responses.

Delegates

The above examples showed instances of control delegates. Control delegates are restricted control plane functions that are installed at sSwitches for control plane efficiency. In the second example above, the path of an IO depends on the workload. `Write` requests can potentially change the location of a subsequent `read`. One way to handle this would be for all requests to be sent by the sSwitch to the controller using the following alternate rules and delegate function:

```
1: Insert (<$VM_1$, w, //$S_1$/X>, (return <IO, Controller>))
2: Insert (<$VM_1$, r, //$S_1$/X>, (return <IO, Controller>))
```

The controller would then serialize and forward them to the appropriate destination. Clearly, this is inefficient, bottlenecking the IO stack at the controller. Instead, the controller uses restricted delegate functions that make control decisions locally at the sSwitches.

This chapter assumes a non-malicious controller, however the design imposes certain functionality restrictions on the delegates to help guard against accidental errors. In particular, delegate functions may only call the APIs in Table 6.2 and may not otherwise keep or create any other state. They may insert or delete rules, but may not rewrite the IO header or IO data. That is important since the IO header contains entries such as source security descriptor that are needed for file access control to work in the rest of the system. These restrictions allow us to consider the delegates as a natural extension of the centralized controller. Simple programming language checks and passing the IO as read-only to the delegate enforce these restrictions.

[4]The request's start offset and data length are part of the IO header.

6.3.5 Consistent rule updates

Forwarding rule updates could lead to instability in the system. This section introduces the notion of consistent rule updates. These updates preserve well-defined storage-specific properties. Similar to networking [119] storage has two different consistency requirements: per-IO and per-flow.

Per-IO consistency. Per-IO consistent updates require that each IO flows either through an old set of rules or an updated set of rules, but not through a stack that is composed of old and new paths. The `Quiesce` and `Drain` calls in the API in Table 6.2 are sufficient to provide per-IO consistent updates.

Per-flow consistency. Many require a stream of IOs to behave consistently. For example, an application might require that a `read` request obtains the data from the latest previous `write` request. In cases where the same source sends both requests, then per-IO consistency also provides per-flow consistency. However, the second request can arrive from a different source, like a second VM in the distributed system. In several basic scenarios, it is sufficient for the centralized controller to serialize forwarding rule updates. The controller disseminates the rules to all sSwitches in two phases. In the first phase, the controller quiesces and drains requests going to the old paths and, in the second phase, the controller updates the forwarding rules.

However, a key challenge are scenarios where delegate functions create new rules. This complicates update consistency since serializing these new rules through the controller is inefficient when rules are created frequently (e.g., for every `write` request). In these cases, control applications attempt to provide all serialization through the sSwitches themselves. They do so as follows. First, they consult the topology map to identify points of serialization along the IO path. The topology map identifies common stages among multiple IO sources on their IO stack. For example, if two clients are reading and writing to the same file X, the control application has the option of inserting two sSwitches with delegate functions close to the two sources to direct both clients' IOs to Y. This option is shown in Figure 6.4(a). The sSwitches would then need to use two-phase commit between themselves to keep rules in sync, as shown in the Figure. This localizes updates to participating sSwitches, thus avoiding the need for the controller to get involved.

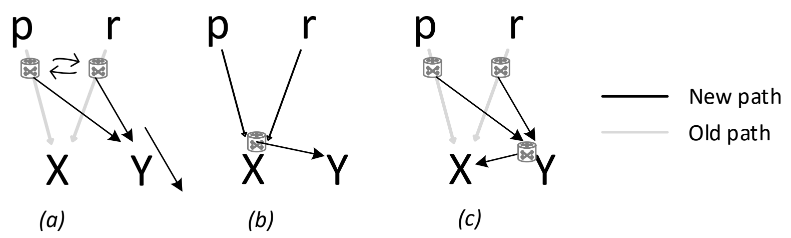


Figure 6.4: Three possible options for placing sSwitches for consistent rule updates. Either can be chosen programmatically at runtime.

A second option would be to insert a single sSwitch close to X (e.g., at the storage server) that forwards IO to Y. This option is shown in Figure 6.4(b). A third option would be to insert an sSwitch at Y that forwards IO back to X if the latest data is not on Y. This type of forwarding rule can be thought of as implementing *backpointers*. Note that two additional sSwitches are needed close to the source to forward all traffic, i.e., reads and writes, to Y, however these sSwitches do not need to perform two-phase commit. The choice between the last two options depends on the workload. If the control application expects that most IO will go to the new file the third option would eliminate an extra network hop.

6.3.6 Fault tolerance and availability

This section analyzes new potential risks on fault tolerance and availability induced by our system. Data continues to be N-way replicated for fault tolerance and its fault tolerance is the same as in the original system.

First, the controller service is new in our architecture. The service can be replicated for availability using standard Paxos-like techniques [76]. If the controller is temporarily unavailable, the implication on the rest of the system is at worst slower performance, but correctness is not affected. For example, IO that matches rules that require transmission to the controller will be blocked until the controller recovers.

Second, our design introduces new metadata in the form of forwarding rules at sSwitches. It is a design goal to maintain all state at sSwitches as soft-state to simplify recovery — also there are cases where sSwitches do not have any local storage available to persist data. The controller itself persists all the forwarding rules before installing them at sSwitches. The controller can choose to replicate the forwarding rules, e.g., using 3-way replication (using storage space available to the controller —either locally or remotely).

However, forwarding rules created at the control delegates pose a challenge because they need persisting. sRoute has two options to address this challenge. The first is for the controller to receive all delegate updates synchronously, ensure they are persisted and then return control to the delegate function. This option involves the controller on the critical path. The second option (the default) is for the delegate rules to be stored with the forwarded IO data. A small header is prepended to each IO containing the updated rule. On sSwitch failure, the controller knows to which servers IO has been forwarded, and recovers all persisted forwarding rules from them.

Third, sSwitches introduce new code along the IO stack, thus increasing its complexity. When sSwitches are implemented in the kernel (see Section 6.4), an sSwitch failure may cause the entire server to fail. We have kept the code footprint of sSwitches small and we plan to investigate software verification techniques in the future to guard against such failures.

6.3.7 Design limitations

In the course of working with sRoute we have identified several current limitations:

- The behaviour and semantics of some file system operations (e.g., file locking) is determined by state stored inside the file system. Because IO routing is done inside the storage stack, above the file system layer, maintaining the semantics of these operations introduces extra complexity to our design. Currently there are two solutions to maintaining the semantics of these operations: i) the state can remain in the file system at the original endpoint, and sSwitches can issue RPCs to query it as IOs flow through the system (naturally, this is not desirable due to the extra communication overhead per IO) or ii) when an sSwitch begins routing a flow, the relevant file system state can be pushed into the sSwitch, and subsequently maintained there, such that the sSwitch can maintain the desired semantics for subsequent incoming IOs.

- IO routing is currently done in the IO stack, without application involvement or application semantics taken into account. While this is sufficient for most of the functionality described here, other functionality may require some amount of application involvement (or knowledge of application semantics) to maintain correctness. For example, performing a backup requires obtaining a consistent snapshot of the application. Storage policies implemented using sRoute that interact with such functionality may require more information about application state, and potentially direct cooperation from the application.

6.4 Implementation

An sSwitch is implemented partly in kernel-level and partly in user-level. The kernel part is written in C and its functionality is limited to partial IO classification through longest prefix matching and forwarding within the same server. The user-level part is written in C#, and implements further sub-file-range classification using hash tables. It also implements forwarding IO to remote servers. An sSwitch is a total of 25 kLOC.

Routing within a server's IO stack. Our implementation makes use of the filter driver architecture in Windows [96]. Each filter driver implements a stage in the kernel and is uniquely identified using an altitude ID in the IO stack. The kernel part of the sSwitch automatically attaches control code to the beginning of each filter driver processing. Bypassing a stage is done by simply returning from the driver early. Going through a stage means going through all the driver code.

Routing across remote servers. To route an IO to an arbitrary remote server's stage, the kernel part of the sSwitch first performs an upcall sending the IO to the user-level part of the sSwitch. That part then transmits the IO to a remote detour location using TCP or RDMA (default) through the SMB file system protocol. On the remote server, an sSwitch intercepts the arriving packet and routes it to a stage within that server.

sSwitch and stage identifiers. An sSwitch is a stage and has the same type of identifier. A stage is identified by a server host name and a driver name. The driver name is a tuple of <device driver name, device name, altitude>. The altitude is an index into the set of drivers or user-level stages attached to a device.

Other implementation details. For the case studies in this chapter, it has been sufficient to inject one sSwitch inside the Hyper-V hypervisor in Windows and another on the IO stack of a remote storage server just above the NTFS file system using file system filter drivers [96]. Specialized functionality is implemented entirely in user-level stages in C#. For example, we have implemented a user-level cache (Section 6.5.3). The controller is also implemented in user-level and communicates with both kernel- and user-level stages through RPCs over TCP.

Routing happens on a per-file basis, at block granularity. Our use cases do not employ any semantic information about the data stored in each block. For control applications that require such information, the functionality would be straightforward to implement, using miniport [94] drivers, instead of filter drivers.

Applications and VMs always run unmodified on our system. However, some applications pass several static hints such as "write through" to the OS using hard-coded flags. The sSwitches intercept `open/create` calls and can change these flags. In particular, for specialized caching (Section 6.5.3) the sSwitches disable OS caching by specifying `Write-through` and `No-buffering` flags. Caching is then implemented through the control application. To avoid namespace conflict with existing files, sRoute stores files in a reserved "sroute-folder" directory on each server. That directory is exposed to the cluster as an SMB share writable by internal processes only.

Implementation limitations. A current limitation of the implementation is that sSwitches cannot intercept individual IO to memory mapped files. However, they can intercept bulk IO that loads a file to memory and writes pages to disk, which is sufficient for most scenarios.

Another current limitation of our implementation is that it does not support byte-range file locking for multiple clients accessing the same file, while performing endpoint routing. The state to support this functionality is kept in the file system, at the original endpoint of the flow. When the endpoint is changed, this state is unavailable. To support this functionality, there are two alternatives, as described in Section 6.3 ¯

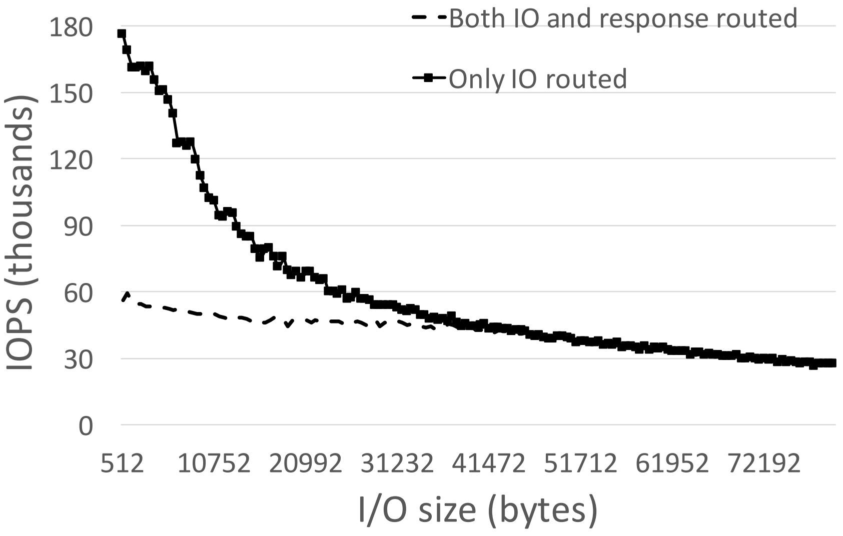


Figure 6.5: Current performance range of an sSwitch.

The performance range of the current implementation of an sSwitch is illustrated in Figure 6.5. This throughput includes passing an IO through both kernel and user-level. Two scenarios are shown. In the "Only IO routed" scenario, each IO has a routing rule but an IO's response is not intercepted by the sSwitch (the response goes straight to the source). In the "Both IO and response routed" scenario both an IO and its response are intercepted by the sSwitch. Intercepting responses is important when the response needs to be routed to a non-default source as well (one of our case studies for caches in Section 6.5.3 requires response routing). Intercepting an IO's response in Windows is costly (due to interrupt handling logic beyond the scope of this chapter) and the performance difference is a result of the OS, not of the sSwitch. Thus the performance range for small IO is between 50,000-180,000 IOPS which makes sSwitches appropriate for an IO stack that uses disk or SSD backends, but not yet a memory-based stack.

6.5 Control applications

This section makes three points. First, we show that a diverse set of control applications can be built on top of IO routing. Thus, we show that the programmable routing abstraction can replace one-off

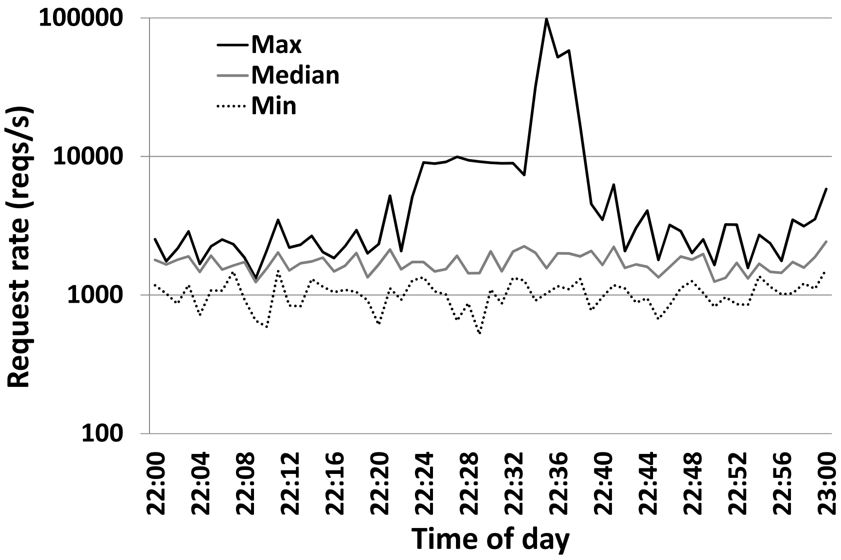


Figure 6.6: Load on three Exchange server volumes showing load imbalances.

hardcoded implementations. We have built and evaluated three control applications implementing tail latency control, replica set control and file cache control. These applications cover each of the detouring types in Table 6.1. Second, we show that tenants benefit significantly from the IO customization provided by the control applications. Third, we evaluate data and control plane performance.

Testbed. The experiments are run on a testbed with 12 servers, each with 16 Intel Xeon 2.4 GHz cores, 384 GB of RAM and Seagate Constellation 2 disks. The servers run Windows Server 2012 R2 operating system and can act as either Hyper-V hypervisors or as storage servers. Each server has a 40 Gbps Mellanox ConnectX-3 NIC supporting RDMA and connected to a Mellanox MSX1036B-1SFR switch.

Workloads. We use three different workloads in this section. The first is TPC-E [152] running over unmodified SQL Server 2012 R2 databases. TPC-E is a transaction processing OLTP workload with small IO sizes. The second workload is a public IO trace from an enterprise Exchange email server [133]. The third workload is IoMeter [62], which we use for controlled micro-benchmarks.

6.5.1 Tail Latency Control

Tail latency in data centers can be orders of magnitude higher than average latency leading to application unresponsiveness [33]. One of the reasons for high tail latency is that IOs often arrive in bursts. Figure 6.6 illustrates this behavior in publicly available Exchange server traces [133], showing traffic to three different volumes of the Exchange trace. The difference in load between the most loaded volume and the least loaded volume is two orders of magnitude and lasts for more than 15 minutes.

Data center providers have load balancing solutions for CPU and network traffic [51]. IO to storage on the other hand is difficult to load balance at short timescales because it is stateful. An IO to an overloaded server S must go to S since it changes state there. The first control application addresses the tail latency problem by temporarily forwarding IOs from loaded servers onto less loaded ones while ensuring that a `read` always accesses the last acknowledged update. This is a type of *endpoint routing*. The functionality provided is similar to Everest [101] but written as a control application that decides when and where to forward to based on global system visibility.

The control application attempts to balance queue sizes at each of the storage servers. To do so, for each storage server, the controller maintains two running averages based on stats it receives[5]: Req_{Avg}, and Req_{Rec}. Req_{Avg} is an exponential moving average over the last hour. Req_{Rec} is an average over a sliding window of one minute, meant to capture the workload's recent request rate. The controller then temporarily forwards IO if:

$Req_{Rec} \quad > \quad \alpha Req_{Avg}$

where α represents the relative increase in request rate that triggers the forwarding. We evaluate the impact of this control application on the Exchange server traces shown in Figure 6.6, but first we show how we map this scenario into forwarding rules.

There are three flows in this experiment. Three different VMs VM_{max}, VM_{min} and VM_{med} on different hypervisors access one of the three volumes in the trace "Max", "Min" and "Median". Each volume is mapped to a VHD file VHD_{max}, VHD_{min} and VHD_{med} residing on three different servers S_{max}, S_{min} and S_{med} respectively. When the controller detects imbalanced load, it forwards write IOs from the VM accessing S_{max} to a temporary file T on server S_{min}:

```
1: $1 :< *, w,\ //S_{max}/VHD_{max} > \rightarrow (F(); return < IO,\ //S_{min}/T >)$
2: $2 :< *, r,\ //S_{max}/VHD_{max} > \rightarrow (return < IO,\ //S_{max}/VHD_{max} >)$
```

Read IOs follow the path to the most up-to-date data, whose location is updated by the delegate function $F()$ as the write IOs flow through the system. We showed how $F()$ updates the rules in Section 6.3.4. Thus, the forwarding rules always point a read to the latest version of the data. If no writes have happened yet, all reads by definition go to the old server VM_{max}.

The control application may also place a specialized stage O in the new path that implements an optional log-structured layout that converts all writes to streaming writes by writing them sequentially to S_{min}. The layout is optional since SSDs already implement it internally and it is most useful for disk-based backends. The control application inserts a rule forwarding IO from the VM first to O (rule 1 below), and another to route from O to S_{min} (rule 2).

```
1: $1 :< *, *,\ //S_{max}/VHD_{max} > \rightarrow (return < IO,\ //S_{min}/O >)$
2: $2 :< O, *,\ //S_{max}/VHD_{max} > \rightarrow (return < IO,\ //S_{min}/T)$
```

Note that in this example data is partitioned across VMs and no VMs share data. Hence, the delegate function in the sSwitch is the only necessary point of metadata serialization in system. This is a simple version of case (a) in Figure 6.4 where sSwitches do not need two-phase commit. The delegate metadata is temporary. When the controller detects that a load spike has ended, it triggers data *reclaim*. All sSwitch rules for writes are changed to point to the original file VHD_{max}. Note that read rules still point to T until new arriving writes overwrite those rules to point to VHD_{max} through their delegate functions. The controller can optionally speed up the reclaim process by actively copying forwarded data to its original location. When the reclaim process ends, all rules can be deleted, the sSwitches and specialized stage removed from the IO stack, since all data resides in and can be accessed again from the original server S_{max}.

We experiment by replaying the Exchange traces using a time-accurate trace replayer on the disk-based testbed. We replay a 30 minute segment of the trace, capturing the peak interval and allowing

[5]The controller uses IOFlow's `getQueueStats` API [147] to gather system-wide statistics for all control applications.

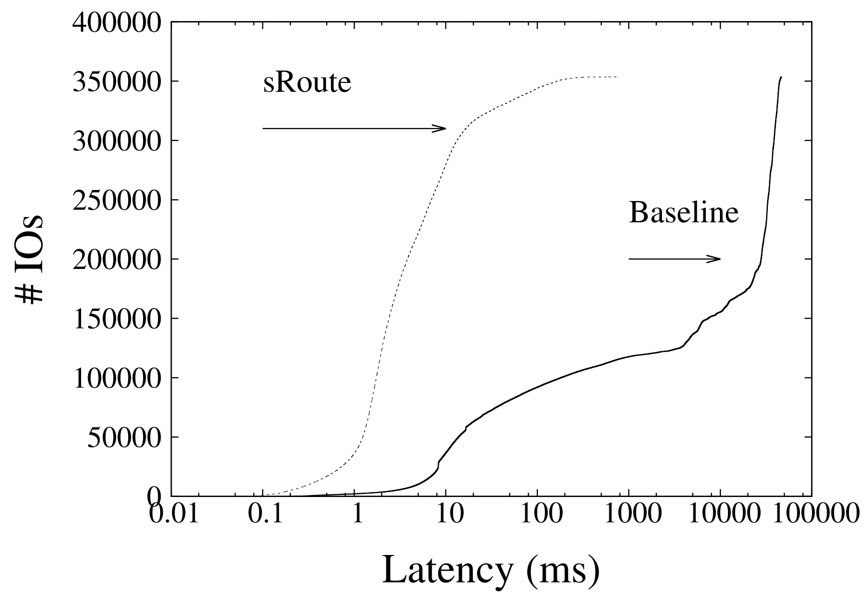


Figure 6.7: CDF of response time for baseline system and with IO routing.

for all forwarded data to be reclaimed. We also employ a log-structured write optimization stage that converts all writes to sequential writes. Figure 6.7 shows the results. IO routing results in two orders of magnitude improvements in tail latency for the flow to S_{max}. The change latency distribution for S_{min} (not shown) is negligible.

Overheads. 2.8 GB of data was forwarded and the delegate functions persisted approximately 100,000 new control plane rules with no noticeable overhead. We experimentally triggered one sSwitch failure, and measured that it took approximately 30 seconds to recover the rules from the storage server. The performance benefit obtained is similar to specialized implementations [101]. The CPU overhead at the controller was less than 1%.

6.5.2 Replica set control

No one replication protocol fits all workloads [2, 145, 78]. Data center services tend to implement one particular choice (e.g, primary-based serialization) and offer it to all workloads passing through the stack (e.g., [17]). One particularly important decision that such an implementation hard-codes is the choice of write-set and read-set for a workload. The write-set specifies the number of servers to contact for a `write` request. The size of the write-set has implications on request latency (a larger set usually means larger latency). The read-set specifies the number of servers to contact for `read` requests. A larger read-set usually leads to higher throughput since multiple servers are read in parallel.

The write- and read-sets need to intersect in certain ways to guarantee a chosen level of consistency. For example, in primary-secondary replication, the intersection of the write- and read-sets contains just the primary server. The primary then writes the data to a write-set containing the secondaries. The request is completed once a subset of the write-set has acknowledged it (the entire write-set by default).

The replica set control application provides a configurable write- and read-set. It uses only *scatter routing* to do so, without any specialized stages. In the next experiment the policy at the control application specifies that if the workload is read-only, then the read-set should be all replicas. However, for correct serialization, if the workload contains writes, all requests must be serialized through the primary, i.e., the read-set should be just the primary replica. In this experiment, the application consists of 10 IoMeters on 10 different hypervisors reading and writing to a 16 GB file using 2-way primary-based replication on the disk testbed. IoMeter uses 4 KB random-access requests and each IoMeter maintains 4 requests outstanding.

The control application monitors the read:write ratio of the workload through IOFlow and when it detects it has been read-only for more than 30 seconds (a configurable parameter) it switches the read-set to be all replicas. To do that, it injects sSwitches at each hypervisor and sets up rules to forward reads to a randomly chosen server S_{rand}. This is done through a control delegate that picks the next server at random. To make the switch between old and new rule the controller firsts `quiesces` writes, then `drains` them. It then inserts the new read-set rule (rule 1):

```
1: $1 :< *, r,\ //S_1/X >\rightarrow (F(); return < IO,\ //S_{rand}/X >)$
2: $2 :< *, w, * >\rightarrow (return < IOHeader,\ Controller >)$
```

The controller is notified of the arrival of any write requests by the rule (2). The controller then proceeds to revert the read-set rule, and restarts the write stream.

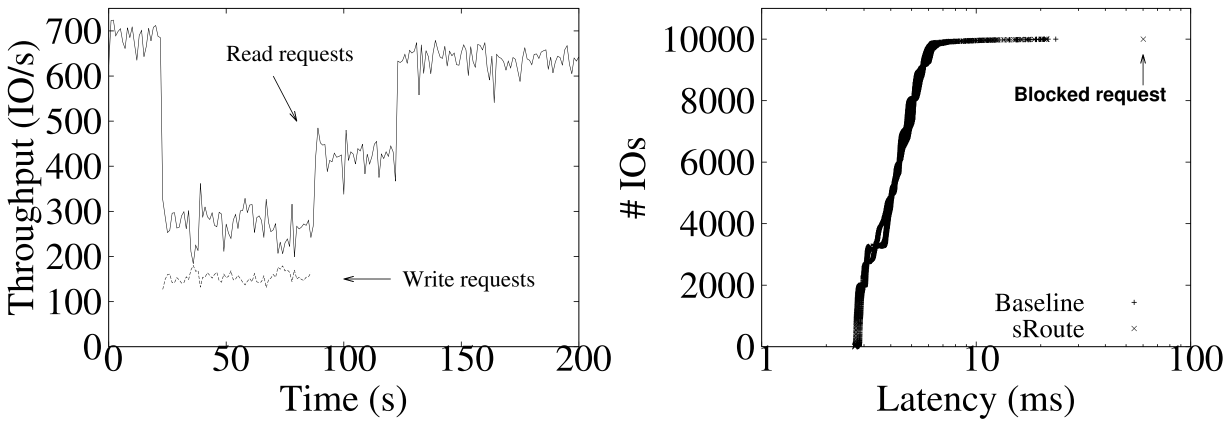


Figure 6.8: Reads benefit from parallelism during read-only phases and the system performs correct serialization during read:write phases (left). The first write needs to block until forwarding rules are changed (right).

Figure 6.8 shows the results. The performance starts high since the workload is in a read-only state. When the first write arrives at time 25, the controller switches the read-set to contain just the primary. In the third phase starting at time 90, writes complete and read performance improves since reads do not contend with writes. In the fourth phase at time 125, the controller switches the read-set to be both replicas, improving read performance by 63% as seen in Figure 6.8(left). The tradeoff is that the first write requests that arrive incur a latency overhead from being temporarily blocked while the write is signalled to the controller, as shown in Figure 6.8(right). Depending on the application performance needs, this latency overhead can be amortized appropriately by increasing the time interval before assuming the workload is read-only. The best-case performance improvement expected is 2x, but the application (IoMeter) has a low number of outstanding requests and does not saturate storage in this example.

Overheads. The control application changes the forwarding rules infrequently at most every 30 seconds. In an unoptimized implementation, a rule change translated to 418 Bytes/flow for updates (40 MB for 100,000 flows). The control application received stats every second using 302 Bytes/flow for statistics (29 MB/s for 100,000 flows). The CPU overhead at the controller is negligible.

6.5.3 File cache control

File caches are important for performance: access to data in the cache is more than 3 orders of magnitude faster than to disks. A well-known problem is that data center tenants today have no control over the location of these caches or their policies [4, 41, 19, 137]. The only abstraction the data center provides to a tenant today is a VMs's memory size. This is inadequate in capturing all the places in the IO stack where memory could be allocated. VMs are inadequate even in providing isolation: an aggressive application within a VM can destroy the cache locality of another application within that VM.

Chapter 5 has explored the programmability of caches on the IO stack, and showed that applications and cloud providers can greatly benefit from the ability to customize cache size, eviction and write policies, as well as explicitly control the placement of data in caches along the IO stack. Such explicit control can be achieved by using filter rules installed in a cache (as shown in Chapter 5). All incoming IO headers are matched against installed filter rules, and an IO is cached if its header matches an installed rule. However, this type of simple control only allows IOs to be cached at some point along their *fixed* path from the application to the storage server. The ability to route IOs to arbitrary locations in the system using sSwitches while maintaining desired consistency semantics allows *disaggregation* of cache memory from the rest of a workload's allocated resources.

This next file cache control application provides several IO stack customizations through *waypoint routing*. We focus on one here: cache isolation among tenants. Cache isolation in this context means that a) the controller determines how much cache each tenant needs and b) the sSwitches isolate one tenant's cache from another's. sRoute controls the path of an IO. It can forward an IO to a particular cache on the data plane. It can also forward an IO to bypass a cache as shown in Figure 6.9.

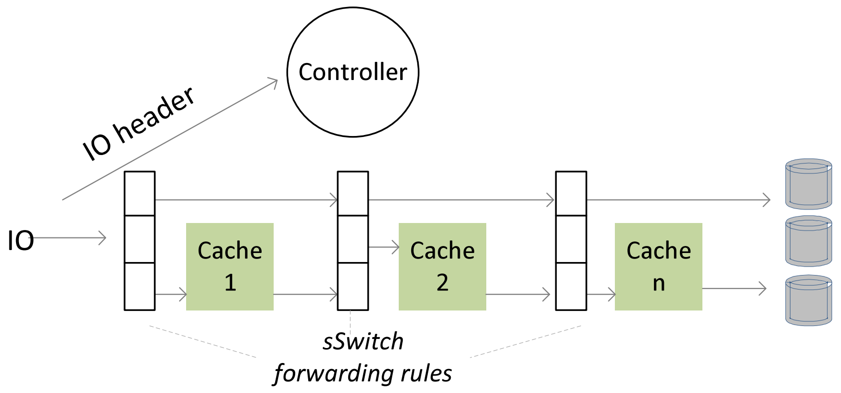


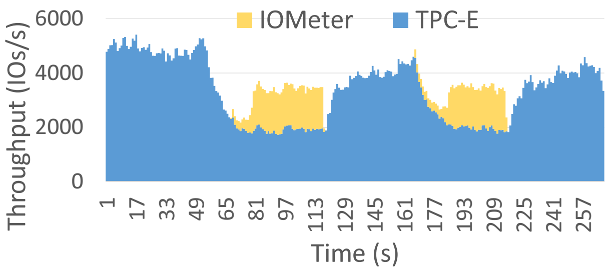
Figure 6.9: Controller sets path of an IO through multiple cache using forwarding rules in sSwitches.

The experiment uses two workloads, TPC-E and IoMeter, competing for a storage server's cache. The storage backend is disks. The TPC-E workload represents queries from an SQL Server database with a footprint of 10 GB running within a VM. IoMeter is a random-access read workload with IO sizes of 512KB. sRoute's policy in this example is to maximize the utilization of the cache with the hit rate measured in terms of IOPS. In the first step, all IO headers are sent to the controller which computes their miss ratio curves using a technique similar to SHARDS [156].

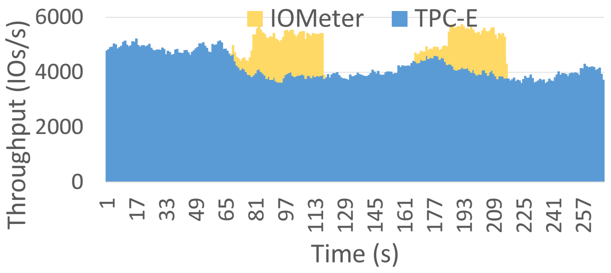
Then, the controller sets up sSwitches so that the IO from IOMeter and from TPC-E go to different caches $C_{IOMeter}$ and C_{TPCE} with sizes provided by SHARDS respectively (the caches reside at the storage server):

1: $< IOMeter, *, * >, (return < IO, C_{IOMeter} >)$

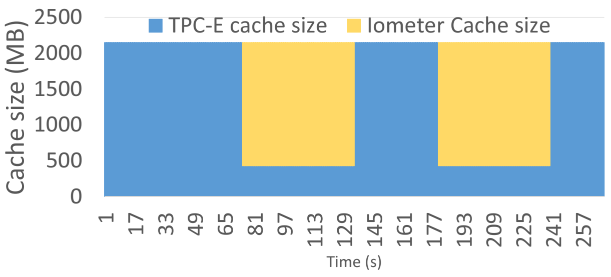
2: $< TPCE, *, * >, (return < IO, C_{TPCE} >)$



(a) Baseline



(b) sRoute



(c) Estimated cache sizes

Figure 6.10: Maximizing hit rate for two tenants with different cache miss curves.

Figure 6.10 shows the performance of TPC-E when competing with two bursts of activity from the IoMeter workload, with and without sRoute. When sRoute is enabled (Figure 6.10(b)), total throughput increases when both workloads run. In contrast, with today's caching (Figure 6.10(a)) total throughput actually drops. This is because IoMeter takes enough cache away from TPC-E to displace its working set out of the cache. With sRoute, total throughput improves by 57% when both workloads run, and TPC-E's performance improves by 2x.

Figure 6.10(c) shows the cache allocations output by our control algorithm when sRoute is enabled. Whenever IoMeter runs, the controller gives it 3/4 of the cache, whereas TPC-E receives 1/4 of the cache, based on their predicted miss ratio curves. This cache allocation leads to each receiving around 40% cache hit ratio. Indeed, the allocation follows the miss ratio curve that denotes what the working set of the TPC-E workload is – after this point diminishing returns can be achieved by providing more cache to this workload. Notice that the controller apportions unused cache to the TPC-E workload 15 seconds after the IoMeter workload goes idle.

Overheads. The control application inserted forwarding rules at the storage server. Rule changes were infrequent (the most frequent was every 30 seconds). The control plane uses approximately 178 Bytes/flow for rule updates (17 MB for 100,000 flows). The control plane subsequently collects statistics from sSwitches and cache stages every control interval (default is 1 second). The statistics are around 456 Bytes/flow (roughly 43 MB for 100,000 flows). We believe these are reasonable control plane overheads. Our current method for generating miss ratio curves (a non-optimized variant of SHARDS) runs offline, and consumes 100% of two cores at the controller.

6.6 Conclusion

This chapter presents sRoute, an architecture that enables an IO routing abstraction, and makes the case that it is useful. We show that many specialized functions on the storage stack can be recast as routing problems. Our hypothesis when we started this work was that, because routing is inherently programmable and dynamic, we could substitute hard-coded one-off implementations with one common routing core. This chapter shows how sRoute can provide unmodified applications with specialized tail latency control, replica set control and achieve file cache isolation, all to substantial benefit.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

The world's computation is moving increasingly into data centers, and digital storage is a crucial part of this move, as we store the large majority of the data we produce as a species in data center systems. The ever-increasing demands on these systems for scale and functionality have put significant strain on the storage stack, leading to a storage stack that is unrealiable and non-performant. This thesis addressed the main barriers to a storage stack that is suitable for today's modern data centers, by improving our understanding of how systems fail in practice and efficient ways of mitigating these failures, as well as introducing new paradigms for how to build and manage the software in the storage stack.

The key results of this dissertation are as follows:

- Errors in main memory can predominantly be attributed to hard errors, which are highly indicative of recurring hardware problems. This knowledge needs to be incorporated into future system designs in order to make systems resilient to memory errors. To this end, page retirement at the OS level was shown to be a very promising solution that does not necessitate hardware changes.
- Raising temperatures in data centers is a promising way of cutting costs spent on their cooling, and reducing their carbon emission impact on the environment. However, doing so entails a performance tradeoff, as many critical system components employ mechanisms to maintain data integrity. These tradeoffs are highly dependent not only on temperature, but particularly on the characteristics of individual workloads.
- The lack of configurability and adaptability of the many software stages across the storage stack can negatively impact application performance, and lead to serious inefficiencies across the stack. Employing a logically-centralized control plane with global visibility to configure and control these software stages allows the storage stack to be vertically specialized and dynamically adapted to workload characteristics, and to different and changing client and cloud provider objectives.
- IO routing is a powerful abstraction that can be used as a programmable primitive to enhance and implement entirely new functionality in the storage stack. Along with global visibility on the control plane, and the ability to configure software stages along the stack, this presents a powerful new paradigm for implementing and managing distributed functionality in the storage stack.

7.2 Future Work

Here, we briefly survey some directions for future work.

- There is more work to be done on making systems resilient to memory errors. In addition to the reactive policies for page retirement we proposed in Chapter 3, further investigation into more proactive policies is also warranted. Our study suggests that current memory scrubbers are not very effective at discovering memory errors, in large part due to their simplistic read-only access patterns. Evidence (both data and anecdotal) sugests that memory scrubbers need to use realistic access patterns more akin to a real application's in order to cause memorry errors to manifest themselves. One approach would be to proactively quarantine suspicious memory areas (migrating their contents to new pages), and scheduling active probing of those areas using realistic access patterns during idle periods, to minimize the impact on system performance. On the mechanism side, current page retirement mechanisms cannot retire kernel memory pages, where the effects of errros are arguably more harmful. Our study shows that kernel pages are more likely to develop errors than application pages, prompting a need for mechanisms that can also mitigate against errors in kernel space.

- Our initial investigations into software-defined storage showed that our approach can be a powerful new paradigm for enhancing and implementing distributed storage functionality. But, there are, of course, several directions for interesting future work.

  1. At the moment, sRoute lacks any verification tools that could help programmers. For example, it is possible to write incorrect control applications that route IOs to arbitrary locations, resulting in data loss. Thus, the routing flexibility is powerful, but unchecked. There are well-known approaches in networking, such as header space analysis [70], that we believe could also apply to storage, but we have not investigated them yet. Similar approaches could also be used to make sure that routing rules from multiple control applications running on the same controller co-exist in the system safely, and that they collectively achieve the correct desired functionality.

  2. Software-defined networking has become a well-established field for several years now. As a research community, we have gained experience with SDN controllers, and following this work also with SDS controllers. It would also be desirable to have a control plane that understands both the network and storage. For example, it is currently possible to get inconsistent end-to-end policies, where the storage controller decides to send data from server A to B, while the network controller decides to block any data from A going to B. Unifying the control plane across resources is an important area for future work.

  3. Another interesting area of exploration relates to handling policies for storage data at rest. Currently, sRoute operates on IO as it is flowing through the system. Once the IO reaches its destination it is considered at rest. It might be advantageous for an sSwitch itself to initiate data movement for data at rest. That would require new forwarding rule types and make an sSwitch more powerful.

  4. It would also be interesting to explore domain-specific languages, to specify and control storage functionality built using the lower-level programmable storage primitives explored in this

thesis. The goal would be to raise the level of abstraction for programmers using our system, and make it easier to specify storage policies and functionality at a higher level, rather than directly controlling the lower-level mechanisms used to implement this functionality.

Bibliography

[1] Soft errors in electronic memory – a white paper. *Tezzaron Semiconductor*.

[2] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. 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 USENIX FAST*, Dec. 2005.

[3] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[4] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP '01, pages 43–56, New York, NY, USA, 2001. ACM.

[5] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, L. N. Bairavasundaram, T. E. Denehy, F. I. Popovici, V. Prabhakaran, and M. Sivathanu. Semantically-smart disk systems: Past, present, and future. *SIGMETRICS Perform. Eval. Rev.*, 33(4):29–35, Mar. 2006.

[6] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 90–105, New York, NY, USA, 2003. ACM.

[7] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of CIDR*, Asilomar, California, 2011.

[8] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[9] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[10] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, Dec. 2007.

[11] C. Belady, A. Rawson, J. Pfleuger, and T. Cader. The Green Grid Data Center Power Efficiency Metrics: PUE & DCiE. Technical report, Green Grid, 2008.

[12] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 267–283, New York, NY, USA, 1995. ACM.

[13] J.-P. Billaud and A. Gulati. hclock: Hierarchical qos for packet scheduling in a hypervisor. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 309–322, New York, NY, USA, 2013. ACM.

[14] D. J. Bradley, R. E. Harper, and S. W. Hunter. Workload-based power management for parallel computer systems. *IBM J.Res.Dev.*, 47:703–718, 2003.

[15] J. Brandon. Going Green In The Data Center: Practical Steps For Your SME To Become More Environmentally Friendly. *Processor*, 29, Sept. 2007.

[16] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, San Jose, CA, 2013. USENIX.

[17] 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 ACM SOSP*, Cascais, Portugal, 2011.

[18] California Energy Commission. Summertime energy-saving tips for businesses. *consumerenergy-center.org/tips/business_summer.html*.

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

[20] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: taking control of the enterprise. In *Proceedings of ACM SIGCOMM*, Kyoto, Japan, 2007.

[21] T. M. Chalfant. Solaris operating system availability features. In *SunBluePrints Online*, 2004.

[22] 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. In *Proceedings of USENIX OSDI*, Seattle, WA, 2006.

[23] Z. Chen, Y. Zhang, Y. Zhou, H. Scott, and B. Schiefer. Empirical evaluation of multi-level buffer cache collaboration for storage systems. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 145–156, New York, NY, USA, 2005. ACM.

[24] G. Chockler, G. Laden, and Y. Vigfusson. Data caching as a cloud service. In *Proceedings of the 4th International Workshop on Large Scale Distributed Systems and Middleware*, LADIS '10, pages 18–21, New York, NY, USA, 2010. ACM.

[25] G. Chockler, G. Laden, and Y. Vigfusson. Design and implementation of caching services in the cloud. *IBM Journal of Research and Development*, 55(6):9:1–9:11, Nov 2011.

[26] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An implementation study of a detection-based adaptive block replacement scheme. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 18–18, Berkeley, CA, USA, 1999. USENIX Association.

[27] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11*, VLDB '85, pages 127–141, Stockholm, Sweden, 1985. VLDB Endowment.

[28] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[29] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proceedings USENIX OSDI*, Hollywood, CA, USA, 2012.

[30] B. Cully, J. Wires, D. 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, Berkeley, CA, USA, 2014. USENIX Association.

[31] H. J. Curnow, B. A. Wichmann, and T. Si. A synthetic benchmark. *The Computer Journal*, 19:43–49, 1976.

[32] A. E. Darling, L. Carey, and W. chun Feng. The design, implementation, and evaluation of mpiblast. In *In Proceedings of ClusterWorld 2003*, 2003.

[33] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, Feb. 2013.

[34] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[35] T. J. Dell. A white paper on the benefits of chipkill-correct ECC for PC server main memory. *IBM Microelectronics*, 1997.

[36] T. J. Dell. System ras implications of dram soft errors. *IBM J. Res. Dev.*, 52(3):307–314, May 2008.

[37] P. E. Dodd. Device simulation of charge collection and single-event upset. *IEEE Nuclear Science*, 43:561–575, 1996.

[38] A. Dragojevic, D. Narayanan, O. Hodson, and M. Castro. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, Seattle, WA, 2014. USENIX Association.

[39] N. El-Sayed, I. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature management in data centers: Why some (might) like it hot. Technical Report TECHNICAL REPORT CSRG-615, University of Toronto, 2012.

[40] N. El-Sayed, I. A. Stefanovici, G. Amvrosiadis, A. A. Hwang, and B. Schroeder. Temperature management in data centers: Why some (might) like it hot. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 163–174, New York, NY, USA, 2012. ACM.

[41] 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 Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[42] A. D. Ferguson, A. Guha, C. Liang, R. Fonseca, and S. Krishnamurthi. Participatory networking: An api for application control of sdns. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 327–338, New York, NY, USA, 2013. ACM.

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

[44] K. Flautner and T. Mudge. Vertigo: Automatic performance-setting for linux. In *Proceedings of the 5th Symposium on Operating Systems Design and implementationCopyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, pages 105–116, Berkeley, CA, USA, 2002. USENIX Association.

[45] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[46] FreeBSD. Freebsd geom storage framework. `http://www.freebsd.org/doc/handbook/`, 2014.

[47] A. Gandhi, M. Harchol-Balter, R. Das, and C. Lefurgy. Optimal power allocation in server farms. In *Proc. of Sigmetrics '09*, 2009.

[48] A. Gara, M. A. Blumrich, D. Chen, G. L.-T. Chiu, P. Coteus, M. E. Giampapa, R. A. Haring, P. Heidelberger, D. Hoenicke, G. V. Kopcsay, T. A. Liebsch, M. Ohmacht, B. D. Steinmacher-Burow, T. Takken, and P. Vranas. Overview of the blue gene/l system architecture. *IBM J. Res. Dev.*, 49(2):195–212, Mar. 2005.

[49] C. Gniady, A. R. Butt, and Y. C. Hu. Program-counter-based pattern classification in buffer caching. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 27–27, Berkeley, CA, USA, 2004. USENIX Association.

[50] R. B. Gramacy, M. K. Warmuth, S. A. Brandt, I. Ari, and I. A. . Adaptive caching by refetching. In *In Advances in Neural Information Processing Systems 15*, pages 1465–1472. MIT Press, 2002.

[51] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of ACM SIGCOMM*, Barcelona, Spain, 2009.

[52] A. Gulati, I. Ahmad, and C. A. Waldspurger. Parda: Proportional allocation of resources for distributed storage access. In *Proccedings of the 7th Conference on File and Storage Technologies*, FAST '09, pages 85–98, Berkeley, CA, USA, 2009. USENIX Association.

[53] A. Gulati, A. Merchant, and P. J. Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 437–450, Berkeley, CA, USA, 2010. USENIX Association.

[54] C. Guo, G. Lu, H. J. Wang, S. Yang, C. Kong, P. Sun, W. Wu, and Y. Zhang. Secondnet: A data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, Co-NEXT '10, pages 15:1–15:12, New York, NY, USA, 2010. ACM.

[55] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[56] K. Harty and D. R. Cheriton. Application-controlled physical memory using external page-cache management. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 187–197, New York, NY, USA, 1992. ACM.

[57] D. Hitz, J. Lau, and M. Malcolm. File system design for an NFS file server appliance. In *Proceedings of USENIX ATC*, San Francisco, California, 1994.

[58] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, 2013. ACM.

[59] A. A. Hwang, I. A. Stefanovici, and B. Schroeder. Cosmic rays don't strike twice: Understanding the nature of dram errors and the implications for system design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122, New York, NY, USA, 2012. ACM.

[60] IDC. The digital universe of opportunities. `https://www.emc.com/collateral/analyst-reports/idc-digital-universe-2014.pdf`, 2014.

[61] Intel. Intel 5400 Chipset Memory Controller Hub (MCH). Technical report, Intel.

[62] Intel Corporation. IoMeter benchmark. `http://www.iometer.org/`, 2014.

[63] M. Isard. Autopilot: Automatic data center management. *SIGOPS Oper. Syst. Rev.*, 41(2):60–67, Apr. 2007.

[64] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 3–14, New York, NY, USA, 2013. ACM.

[65] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, C. Kim, and A. Greenberg. Eyeq: Practical network performance isolation at the edge. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 297–312, Berkeley, CA, USA, 2013. USENIX Association.

[66] I. journal of Research and D. staff. Overview of the ibm blue gene/p project. *IBM J. Res. Dev.*, 52(1/2):199–220, Jan. 2008.

[67] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, SOSP '97, pages 52–65, New York, NY, USA, 1997. ACM.

[68] J. M. Kaplan, W. Forrest, and N. Kindler. Revolutionizing data center energy efficiency. Technical report, McKinsey & Company, July 2008.

[69] J. Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, Oct. 1997.

[70] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proceedings of NSDI*, Lombard, IL, 2013.

[71] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 9–9, Berkeley, CA, USA, 2000. USENIX Association.

[72] H. Kobayashi, K. Shiraishi, H. Tsuchiya, H. Usuki, Y. Nagai, and K. Takahisa. Evaluation of lsi soft errors induced by terrestrial cosmic rays and alpha particles. Technical report, Sony corporation and RCNP Osaka University, 2001.

[73] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *Proceedings of USENIX OSDI*, Vancouver, BC, Canada, 2010.

[74] K. Krueger, D. Loftesness, A. Vahdat, and T. Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings of ACM OOPSLA*, Washington, D.C., USA, 1993.

[75] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, Apr. 2010.

[76] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[77] Lawrence Berkeley National Labs. Benchmarking Data Centers. `http://hightech.lbl.gov/benchmarking-dc.html`, December 2007.

[78] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of USENIX OSDI*, OSDI'12, Hollywood, CA, USA, 2012.

[79] X. Li, A. Aboulnaga, K. Salem, A. Sachedina, and S. Gao. Second-tier cache management using write hints. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4*, FAST'05, pages 9–9, Berkeley, CA, USA, 2005. USENIX Association.

[80] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[81] X. Li, K. Shen, M. C. Huang, and L. Chu. A memory soft error measurement on production systems. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, ATC'07, pages 21:1–21:6, Berkeley, CA, USA, 2007. USENIX Association.

[82] M. Lin, A. Wierman, L. L. H. Andrew, and E. Thereska. Dynamic right-sizing for power-proportional data centers. *IEEE/ACM Trans. Netw.*, 21(5):1378–1391, Oct. 2013.

[83] S. Lin and D. J. Costello. *Error Control Coding, Second Edition*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004.

[84] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[85] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan. 1979.

[86] J. D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Comp. Soc. TCCA Newsletter*, pages 19–25, Dec. 1995.

[87] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, Mar. 2008.

[88] L. McVoy and C. Staelin. Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA, 1996. USENIX Association.

[89] 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, Berkeley, CA, USA, 2003. USENIX Association.

[90] A. Merkel and F. Bellosa. Balancing power consumption in multiprocessor systems. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 403–414, New York, NY, USA, 2006. ACM.

[91] M. Mesnier, F. Chen, T. Luo, and J. B. Akers. Differentiated storage services. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 57–70, New York, NY, USA, 2011. ACM.

[92] Microsoft. Microsoft research storage toolkit. `http://research.microsoft.com/en-us/downloads/230b8ad4-3340-4a87-8ef0-cf92b376db86/`.

[93] Microsoft. Virtual hard disk performance. `http://download.microsoft.com/download/0/7/7/0778C0BB-5281-4390-92CD-EC138A18F2F9/WS08_R2_VHD_Performance_WhitePaper.docx`.

[94] Microsoft Corporation. Minidrivers, miniport drivers, and driver pairs. `https://msdn.microsoft.com/en-us/library/windows/hardware/hh439643.aspx`.

[95] Microsoft Corporation. File system minifilter allocated altitudes (MSDN). `http://msdn.microsoft.com/`, 2013.

[96] Microsoft Corporation. File system minifilter drivers (MSDN). `https://msdn.microsoft.com/en-us/windows/hardware/drivers/ifs/file-system-minifilter-drivers`, 2014.

[97] R. Miftakhutdinov, E. Ebrahimi, and Y. N. Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 155–165, Washington, DC, USA, 2012. IEEE Computer Society.

[98] G. Miłós, D. G. Murray, S. Hand, and M. A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, pages 1–1, Berkeley, CA, USA, 2009. USENIX Association.

[99] B. Murphy. Automating software failure reporting. *Queue*, 2(8):42–48, Nov. 2004.

[100] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *Trans. Storage*, 4(3):10:1–10:23, Nov. 2008.

[101] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. 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, Berkeley, CA, USA, 2008. USENIX Association.

[102] R. Nathuji and K. Schwan. Virtualpower: Coordinated power management in virtualized enterprise systems. *SIGOPS Oper. Syst. Rev.*, 41(6):265–278, Oct. 2007.

[103] D. C. Niemi. Unixbench. `http://www.tux.org/pub/tux/niemi/unixbench/`.

[104] E. Normand. Single event upset at ground level. *IEEE Transaction on Nuclear Sciences*, 6(43):2742–2750, 1996.

[105] T. J. O'Gorman, J. M. Ross, A. H. Taber, J. F. Ziegler, H. P. Muhlfeld, C. J. Montrose, H. W. Curtis, and J. L. Walsh. Field testing for cosmic ray soft errors in semiconductor memories. *IBM J. Res. Dev.*, 40(1):41–50, Jan. 1996.

[106] M. Ohmacht, R. A. Bergamaschi, S. Bhattacharya, A. Gara, M. E. Giampapa, B. Gopalsamy, R. A. Haring, D. Hoenicke, D. J. Krolak, J. A. Marcella, B. J. Nathanson, V. Salapura, and M. E. Wazlowski. Blue gene/l compute chip: Memory and ethernet subsystem. *IBM J. Res. Dev.*, 49(2):255–264, Mar. 2005.

[107] Oracle. Oracle Solaris ZFS administration guide. `http://docs.oracle.com/cd/E19253-01/819-5461/index.html`.

[108] S. V. Patankar. Airflow and cooling in a data center. *Journal of Heat transfer*, 132(7):073001, 2010.

[109] C. D. Patel, C. E. Bash, C. Belady, L. Stahl, and D. Sullivan. Computational fluid dynamics modeling of high compute density data centers to assure system inlet air specifications. In *Proceedings of IPACK*, volume 1, pages 8–13, 2001.

[110] C. D. Patel, C. E. Bash, R. Sharma, and M. Beitelmal. Smart cooling of data centers. In *Proc. of IPACK*, 2003.

[111] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[112] E. Pinheiro, R. Bianchini, E. V. Carrera, and T. Heath. Load balancing and unbalancing for power and performance in cluster-based systems. In *Proc. of Workshop on Compilers and Operating Systems for Low Power (COLP)*, 2001.

[113] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, pages 2–2, Berkeley, CA, USA, 2007. USENIX Association.

[114] S. J. Plimpton, R. Brightwell, C. T. Vaughan, K. D. Underwood, and M. Davis. A simple synchronous distributed-memory algorithm for the hpcc randomaccess benchmark. In *CLUSTER*. IEEE Computer Society, 2006.

[115] L. Popa, G. Kumar, M. Chowdhury, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. Faircloud: Sharing the network in cloud computing. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 187–198, New York, NY, USA, 2012. ACM.

[116] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM.

[117] K. Rajamani and C. Lefurgy. On evaluating request-distribution schemes for saving energy in server clusters. In *Proc. of the IEEE ISPASS*, 2003.

[118] R. V. Rein. BadRAM: Linux kernel support for broken RAM modules.

[119] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA, 2012. ACM.

[120] Rich Miller. Google: Raise your data center temperature. *http://www.datacenterknowledge.com/archives/2008/10/14/google-raise-your-data-center- temperature/*, 2008.

[121] A. Riska and E. Riedel. Idle read after write: Iraw. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 43–56, Berkeley, CA, USA, 2008. USENIX Association.

[122] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[123] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 28:1–28:14, New York, NY, USA, 2014. ACM.

[124] B. Schroeder and G. A. Gibson. A large-scale study of failures in high-performance computing systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 249–258, Washington, DC, USA, 2006. IEEE Computer Society.

[125] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies*, FAST '07, Berkeley, CA, USA, 2007. USENIX Association.

[126] B. Schroeder, E. Pinheiro, and W.-D. Weber. Dram errors in the wild: A large-scale field study. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 193–204, New York, NY, USA, 2009. ACM.

[127] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 213–227, New York, NY, USA, 1996. ACM.

[128] R. Sharma, C. Bash, C. Patel, R. Friedrich, and J. Chase. Balance of power: dynamic thermal management for internet data centers. *IEEE Internet Computing*, 9(1):42–49, 2005.

[129] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: Network processing as a cloud service. In *Proceedings of the ACM SIGCOMM*, Helsinki, Finland, 2012.

[130] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the data center network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 309–322, Berkeley, CA, USA, 2011. USENIX Association.

[131] D. Shue, M. J. Freedman, and A. Shaikh. Performance isolation and fairness for multi-tenant cloud storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 349–362, Berkeley, CA, USA, 2012. USENIX Association.

[132] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[133] SNIA. Exchange server traces. `http://iotta.snia.org/traces/130`.

[134] G. Soundararajan, J. Chen, M. A. Sharaf, and C. Amza. Dynamic partitioning of the cache hierarchy in shared data centers. *Proc. VLDB Endow.*, 1(1):635–646, Aug. 2008.

[135] C. Staelin. Lmbench3: Measuring scalability. Technical report, HP Laboratories Israel, 2002.

[136] I. Stefanovici, B. Schroeder, G. O'Shea, and E. Thereska. sroute: Treating the storage stack like a network. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 197–212, Santa Clara, CA, Feb. 2016. USENIX Association.

[137] I. Stefanovici, E. Thereska, G. O'Shea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 174–181, New York, NY, USA, 2015. ACM.

[138] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Trans. Comput.*, 41(9):1054–1068, Sept. 1992.

[139] R. F. Sullivan. Alternating Cold and Hot Aisles Provides More Reliable Cooling for Server Farms. In *Uptime Institute*, 2000.

[140] T10 Technical Committee. SCSI Block Commands – 3, Rev.25. Work. Draft T10/1799-D, ANSI INCITS.

[141] T13 Technical Committee. ATA 8 - ATA/ATAPI Command Set, Rev.4a. Work. Draft T13/1699-D, ANSI INCITS.

[142] D. Tang, P. Carruthers, Z. Totari, and M. W. Shapiro. Assessment of the effect of memory page retirement on system ras against hardware faults. In *Proceedings of the International Conference on Dependable Systems and Networks*, DSN '06, pages 365–370, Washington, DC, USA, 2006. IEEE Computer Society.

[143] H. H. Tang. Semm-2: A new generation of single-event-effect modeling tools. *IBM J. Res. Dev.*, 52(3):233–244, May 2008.

[144] H. H. K. Tang, C. E. Murray, G. Fiorenza, K. P. Rodbell, M. S. Gordon, and D. F. Heidel. New simulation methodology for effects of radiation in semiconductor chip structures. *IBM J. Res. Dev.*, 52(3):245–253, May 2008.

[145] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of ACM SOSP*, Farmington, Pennsylvania, 2013.

[146] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of ACM SOSP*, Copper Mountain, Colorado, United States, 1995.

[147] E. Thereska, H. Ballani, G. O'Shea, T. Karagiannis, A. Rowstron, T. Talpey, R. Black, and T. Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 182–196, New York, NY, USA, 2013. ACM.

[148] E. Thereska, A. Donnelly, and D. Narayanan. Sierra: practical power-proportionality for data center storage. In *Proceedings of Eurosys*, pages 169–182, Salzburg, Austria, 2011.

[149] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for internet data transfer. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association.

[150] Transaction Processing Performance Council. TPC Benchmark C - Rev. 5.11. Standard, Feb. 2010.

[151] Transaction Processing Performance Council. TPC Benchmark H - Rev. 2.14.2. Standard, June 2011.

[152] Transaction Processing Performance Council. TPC Benchmark E - Rev. 1.14.0. Standard, June 2014.

[153] USENIX. The computer failure data repository (CFDR). `https://www.usenix.org/cfdr`.

[154] J. W. VanGilder and R. R. Schmidt. Airflow uniformity through perforated tiles in a raised-floor data center. In *ASME 2005 Pacific Rim Technical Conference and Exhibition on Integration and Packaging of MEMS, NEMS, and Electronic Systems collocated with the ASME 2005 Heat Transfer Summer Conference*, pages 493–501. American Society of Mechanical Engineers, 2005.

[155] C. A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, Dec. 2002.

[156] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, Santa Clara, CA, 2015. USENIX Association.

[157] R. P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, Oct. 1984.

[158] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, Broomfield, CO, Oct. 2014. USENIX Association.

[159] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of USENIX ATC*, Monterey, California, 2002.

[160] H. Yan, D. A. Maltz, T. S. E. Ng, H. Gogineni, H. Zhang, and Z. Cai. Tesseract: a 4D network control plane. In *Proceedings of USENIX NSDI*, Cambridge, MA, 2007.

[161] D. H. Yoon and M. Erez. Virtualized and flexible ecc for main memory. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 397–408, New York, NY, USA, 2010. ACM.

[162] J. F. Ziegler. Terrestrial cosmic rays. *IBM J. Res. Dev.*, 40(1):19–39, Jan. 1996.

[163] J. F. Ziegler, H. W. Curtis, H. P. Muhlfeld, C. J. Montrose, and B. Chin. Ibm experiments in soft fails in computer electronics (1978&ndash;1994). *IBM J. Res. Dev.*, 40(1):3–18, Jan. 1996.

[164] J. F. Ziegler and W. A. Lanford. Effect of Cosmic Rays on Computer Memories. In *Science*, volume 206, pages 776–788, 1979.

[165] J. F. Ziegler, M. E. Nelson, J. D. Shell, R. J. Peterson, C. J. Gelderloos, H. P. Muhlfeld, and C. J. Montrose. Cosmic ray soft error rates of 16-Mb DRAM memory chips. *Solid-State Circuits, IEEE Journal of*, 33(2):246–252, 1998.