OPERATING SYSTEMS FOR FAR OUT MEMORIES

DANIEL BITTMAN

June 2023 – v. 1.1

COLOPHON

The author battled LaTeX for typesetting using an unabashedly strong influence from Aaron Turon's dissertation[1] as inspiration for formatting. The document uses a modified `classicthesis` by André Miede. A variety of fonts are used, but there is no Arial to be found.

[1] http://aturon.github.io/academic/turon-thesis.pdf

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

NVM	Non-volatile Memory
KVS	Key-Value Store
DSM	Distributed Shared Memory

ABSTRACT

Operating systems are built and designed around two driving forces: the capabilities of hardware, and the demands of software. Yet traditional[2] operating systems and programming models have inertia, resulting in interfaces for new hardware following the designs of existing interfaces. As a result, programmers are limited in their ability to express the important parts of their programs due to the layers of compatibility and overhead thrust upon them, despite their persistent demands for higher throughput and lower latency. Operating system abstractions must evolve into the modern day. Merely relying on decades old abstractions and incremental change will relegate novel hardware of the last decade to a fate of access via interfaces designed for tape and spinning rust.

We stand before an opportunity to study how a confluence of trends may shift programming models away from a traditional, process-centric view point towards a *data-centric* one, in which *data* is the primary citizen of the system. This opportunity arises from trends in hardware that directly impact how we view the data access path and the responsibilities of the operating system and the kernel. The increasing speed of interconnect technologies draws computing nodes closer together in latency space, increasing the efficiency and useability of shared memory. Persistence, traditionally trapped low in the memory hierarchy, is leaking upward, as access speeds for persistent devices increase[3]. As a result, the overhead of the traditional kernel-driven data access path begins to dominate the cost of accessing persistent data. Finally, the increasing heterogeneity and disaggregation of compute and memory devices demands increased data and compute mobility, as software demands continued scalability, distribution, and raw speed.

This dissertation presents a new, data-centric operating system and programming model designed around the trends above. The data-centric approach reframes the goals of the operating system and enables us to re-imagine classic systems programming techniques into a model that *facilitates* data sharing instead of hindering it. For example, classical systems programming models and techniques tend to involve significant complexity and overhead in dealing with data persistence and sharing, such as expensive coarse-grained persistence operations, rigid RPC data models, and serialization. In contrast, our data-centric approach gets the kernel out of the way of the data access path, makes data mobile through invariant references whose meaning does not change depending on address space or machine[4], and thus removes the need for serialization.

[2] Read: old.

[3] This includes new technologies like byte-addressible non-volatile memory DIMMs, but also improvements in SSD performance and interfaces.

[4] In contrast to virtual memory, whose references only have meaning inside a single address space.

We will cover the motivation and hardware trends that lead to our design, define a design space based on those trends, and finally discuss Twizzler, a point in that design space that exemplifies the ideals we will discuss. We will evaluate Twizzler with case-studies that demonstrate system behavior, and efficacy and useability of our programming models, by building several new pieces of software for Twizzler. These, along with ported larger applications, will be used to demonstate the performance of Twizzler and its programming model, often showing performance increase just due to simplification of software layers.

For
Ellen,
Steve,
and Christina.

PUBLICATIONS

[1] Daniel Bittman, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. “A Tale of Two Abstractions: The Case for Object Space.” In: *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotstorage19/presentation/bittman.

[2] Daniel Bittman, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. “Optimizing Systems for Byte-Addressable NVM by Reducing Bit Flipping.” In: *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST ’19)*. Feb. 2019.

[3] Daniel Bittman, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. “The Flipside: A Bit Flip Saved is Power and Lifetime Earned.” In: *;login:* 44.2 (June 2019), pp. 27–31.

[4] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. “Twizzler: A Data-Centric OS for Non-Volatile Memory.” In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 65–80. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/bittman.

[5] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. “Twizzler: A Data-Centric OS for Non-Volatile Memory.” In: *ACM Transactions on Storage* 17.2 (June 2021). ISSN: 1553-3077. DOI: 10.1145/3454129. URL: https://doi.org/10.1145/3454129.

[6] Daniel Bittman, Peter Alvaro, and Ethan L. Miller. “A Persistent Problem: Managing Pointers in NVM.” In: *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS ’19)*. Oct. 2019, pp. 30–37.

[7] Daniel Bittman, Matthew Bryson, Yuanjiang Ni, Arjun Govindjee, Isaak Cherdak, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. *Twizzler: An Operating System for Next-Generation Memory Hierarchies*. Tech. rep. UCSC-SSRC-17-01. University of California, Santa Cruz, Dec. 2017.

[8] Daniel Bittman, Matthew Gray, Justin Raizes, Sinjoni Mukhopadhyay, Matt Bryson, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. “Designing Data Structures to Minimize Bit Flips on NVM.” In: *Proceedings of the 7th IEEE Non-Volatile Memory Sys-*

tems and Applications Symposium (NVMSA 2018). Aug. 2018. URL: http://www.ssrc.ucsc.edu/bittman-nvmsa18.pdf.

[9] Daniel Bittman, Robert Soule, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. "Don't Let RPCs Constrain Your API." In: *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets '21)*. Nov. 2021.

ACKNOWLEDGMENTS

None of this work could have been completed without tremendous help, guidance, support, and love. I'd like to say a few words about all the people who came together to realize this thesis.

My partner, Emily Zierdt Smith, has been my biggest supporter throughout my studies, and has stood by me through good times and bad during my PhD. She has given me feedback, helped me formulate and make sense of my thoughts and myself, and, with great patience and love, has helped me along the way through the marathon that is graduate school. Perhaps one day I can repay her kindness and patience for these graduate school years, but that may take a lifetime.

My PhD would have been over before it began if not for the guidance of the members comprising my dissertation committee, who, in no small way, provided me the support and the environment I needed to succeed. In no particular order, I want to profusely thank:

- Peter Alvaro, *whose blue-sky insights have shaped this work and story like no other.*

- Ethan Miller, *with his concrete and meticulous approach to systems engineering.*

- Darrell Long, *whose kindness is matched equally well with invaluable critical feedback.*

- Pankaj Mehra, *whose propensity to probe at the difficult questions of research always leads to better understanding.*

The above list is in no way meant to imply that these qualities are completely unique to each person; on the contrary, I look up to these people precisely because they all share these vital qualities in a mentor. They have all provided me guidance and training for analyzing problems, building mental models, probing those models, designing experiments, running and managing them, collecting and processing data, turning that into new knowledge, and of course, communicating that knowledge. And, perhaps most important of all, zooming out and looking at the big picture. I can only hope to emulate their brilliance going forward.

I will never forget the unending support from my lab mates, in no particular order, Dev Purandare, Peter Wilcox, Michael Usher, Achilles Benetopoulos, Esteban Ramos, Matt Bryson, Staunton Sample, Yan Li, Lincoln Thurlow, Oceane Bel, Ken Chang, Sinjoni Mukhopadhyay, Jim Hughes, James Byron, and Yuanjiang Ni, among others. My work would not have come together if not for whiteboarding sessions, feedback, and discussions with these fine folks. I want to explicitly thank DJ Capelis, who mentored me during the last few years of undergrad and helped induct me into a graduate lab and give me a taste for research from which I never recovered. I will miss you all, but don't think that you'll never see me again.

I'd additionally like to thank Robert Soulé, Avi Silberschatz, Shel Finkelstein, Andrew Quinn, Ike Nassi, Heiner Litz, Pat Heland, Bruce Lindsay, and Ahmed Amer for their feedback and discussions. Having the ear of these brilliant people, but more importantly listening to their insights, has been invaluable.

Unfortunately, my time in graduate school will forever be tied to the COVID-19 pandemic. I am forever grateful to my labmates for working hard to maintain morale. I will cherish our remote tea times. My heart goes out to all graduate students who suffered in academia during these last few years.

Finally, all of my friends and family who stuck with me and supported me over the years. You are all amazing, and I love each and every one of you. My parents and sister never ceased in their unending support, and I cannot thank them enough. This dissertation is dedicated to them, from the bottom of my heart.

Thank you, all. ♡

Daniel Bittman

"I wish it need not have happened in my time," said Frodo. "So do I," said Gandalf, "and so do all who live to see such times. But that is not for them to decide. All we have to decide is what to do with the time that is given us."

—*The Lord of the Rings*, J.R.R. Tolkien

FOREWORD

BASHIR: Out of all the stories you told me—which ones were true, and which ones weren't?
GARAK: My dear doctor! They're all *true.*
BASHIR: Even the lies?
GARAK: Especially *the lies.*

—Star Trek: Deep Space Nine

This dissertation presents a narrative. That narrative is, at time of writing, the best understanding of the work that I have. Any researcher knows that one doesn't start a research expedition with a fully formed understanding of a narrative in-mind; instead, that narrative grows and changes dramatically with each success, but more importantly with each failure. And while I'd like the narrative—the *story*—of the work herein to contain not just the sapling that is the product of my time at UCSC, but also the dead branches that I've pruned along the way, such story telling would distract from the scientific communication of this work. But I'd like to expose a sample of that more accurate, but no less "true", story here.

Audiences know what to expect, and that is all that they are prepared to believe in.

—*Rosencrantz and Guildenstern Are Dead*, Tom Stoppard

One might well question the logic behind prefacing this disseratation with a refutation of its presented narrative. However, I think it's important to consider what effect a work may have upon a reader. In particular, if even one student reads these chapters and comes away with a belief that the scientific process (and, more specifically, the Ph.D. process) is linear and *not* an experience of one feeling their way through a dark labyrinth, then I will have been negligent in my duty to help future students and avoid harming them.

All this happened, more or less.

—*Slaughterhouse-Five*, Kurt Vonnegut

Twizzler started as a combination of two things: my love for hacking kernels, and an idea for a new OS focused on the exciting new technology of byte-addressable non-volatile memory (NVM). It has since grown well beyond that limited scope, but at the time, that limitation allowed me to get started. We hadn't yet figured out the "big picture" words to use to describe what we were trying to do, instead we presented the work as a straight-forward design to improve performance for applications on NVM. It seemed like the perfect time, with Intel releasing 3D-Xpoint memory, and the interest in NVM exploding. All I had to do was build an operating system to manifest the designs we'd been whiteboarding.

I started by prototyping the ideas inside FreeBSD, modifying the kernel to act like we wanted it to. This was largely a dead end, since we rapidly ran into walls trying to force a Unix kernel to be something it wasn't. The FreeBSD prototype was superceded with a custom kernel and grew into the main Twizzler operating system that is presented herein. It needed to be reworked and redesinged several times, each time leading to me pulling long nights programming and writing. But, in the end, the system worked, despite us not yet having the right words to describe it.

Since I mentioned failure earlier, let's talk about publications. It took us *years* to get Twizzler published in a conference. Along the way we had two workshop publications, but the rejection notifications from conferences were, shall we say, starting to get to me. Fortunately, I had supportive professors who showed me their "co-CVs", containing lists of rejected publications. I started keeping one as well. This is all to say—if you're a student, and you are struggling to publish: we have all been there[5], and (despite how some program committees behave) we are rooting for you. Maybe one day our academic publishing system will function well.

[5] Anyone who claims to know, in full, what they are doing, is a *liar*.

I've also had blessings in disguise. As I said earlier, the initial writings on Twizzler framed the work nearly exclusively around NVM and performance ("getting the kernel out of the way"). Looming on the horizon, however, was the growing certainty that NVM (or, at least, 3D-Xpoint) would *not* soon deliver on its promises. Looking back, we got lucky. We were forced to rethink the narrative, to look at the forest instead of the trees, and I think the work is much stronger as a result. We were able to generalize. In fact, one of the benefits to the design presented in Chapter 6 is that, while we built it for NVM at the start, it is actually quite generalizable to larger contexts, and so much of the technical design work for Twizzler wasn't lost during this rethinking of the narrative.

The research process is labyrinthian, and never ending. The snapshot of work presented here is distilled into a story that is, in my opinion, straight-forward and compelling. But the road to get here was long, filled with pot holes, dead ends, and complete reimaginings. I hope you enjoy reading it as much as I have enjoyed making this crazy idea a reality.

♡ ♡ ♡

Daniel Bittman

Part I

PRELUDE

It's the best possible time to be alive, when almost everything you thought you knew is wrong.

—Valentine, *Arcadia*, Tom Stoppard

1 INTRODUCTION

Let's design and build a new operating system.

"Sometimes I find the best way of getting from one place to another is simply to erase everything and begin again."

—*The Phantom Tollbooth*, Norton Juster

THE MOTIVATION

The confluence of several hardware trends—persistent memory moving up the memory hierarchy, faster interconnect technologies, and increasing heterogeneity of compute and memory—demands a fundamental shift in how we view applications' relationships with hardware and data. As persistence gets closer to compute in latency space, the line separating the traditional two-tier memory hierarchy of fast, volatile memory and slow, persistent storage begins to blur. Meanwhile, interconnect technologies are improving, causing separate computing nodes to be drawn closer to each other in latency space, allowing them to more efficiently share memory. Computing resources are spreading out, as we race to place computing units in devices and near memory, and as we start seeing different kinds of physical memory on the bus, the traditional host-centric and process-centric models of programming give way to models that better express the increasing demands for concurrency and parallelism between devices and computers.

Software both drives some of these trends, as it demands increasing throughput and lower latency when processing data, but is also affected by the trends, or more specifically, often limited in expressivity to what *abstractions* are presented to software by the operating system[6]. The primary goal of a program is to access and operate on data and any additional required work that an application must perform to enable that goal is overhead, both in terms of performance (latency or throughput) but also *complexity*. Applications must routinely pay significant overhead costs that ultimately source from the abstractions and programming model enabled by the operating system, which is in turn sourced from a model of hardware decades old. Traditional operating system abstractions and interfaces do not adequately reflect current hardware, the direction hardware is heading, nor the demands of software upon those interfaces.

[6] OS abstractions, a favorite punching bag!

THE OPPORTUNITY

Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.

—Robert Frost

We have an opportunity to examine hardware trends and evolve operating system design to make the best possible use of new technologies and trends. Mere incremental change will not enable the dramatic improvements in performance and complexity heralded by these trends, just as SSDs did not reach their full potential until they transcended the disk paradigm[7]. Instead, we will discuss and examine a "clean-slate" approach to operating system design that avoids trying to mold new hardware into some backwards-compatible existing box[8]. Such an approach will necessitate examining past research and reconsidering previously impractical ideas[9] while extending those ideas for modern software, hardware, and languages. Simultaneously, we must design new abstractions and interfaces that expose a programming model that allows applications to center around the data they are accessing while not requiring them to twist into knots trying to best utilize modern hardware.

[7] Arguably, they still haven't. SSDs were, for a long time, treated as "fast disks", to the detriment of our storage stacks.

[8] Imagine if we tried to access, say, persistent RAM with interfaces designed for magnetic tape! *Stares straight into the camera.*

[9] Knowledge derives from experience of the world, but what if that world were to change?

Our focus will be on a *data-centric* approach, in which *data* is the primary citizen instead of the process. As we will see, this framing around data forces us to reconsider and reimagine classic systems programming tropes like explicit, coarse-grained persistence barriers, call-by-small-value RPC, shared memory, serialization, and in-memory data structures that cannot escape the bounds of a single process[10]. We will define a design space for data-centric operating systems that centers around in-memory data in a global address space that can be shared across both space and time, whose lifetime is disconnected from ephemera like processes, nodes, and virtual address spaces, and which can be operated on, persisted, and shared without the overhead of operating system involvement in the data access path[11]. In a world where in-memory data can last forever and move across processes and nodes, the context required to manipulate that data is best coupled with the *data* rather than ephemeral constructs. Data has always been the focus of programming; it's time our operating system abstractions adequately capture this simple fact.

[10] See Chapter 3.

[11] See Chapter 4.

THE IMPLEMENTATION

The principal hypothesis of this dissertation is that the data-centric model for designing operating system abstractions is not only viable, but demanded both by software and hardware trends. Examining this hypothesis will require answering a number of questions about the design space, the practicality of any implementation of our ideas, and empirical measurements of that implementation. To study data-centric operating system design, we have built *Twizzler*, an operating system that embodies the ideas presented herein.

Twizzler consists of a standalone kernel built from scratch, a set of userspace libraries for programming memory, and a set of applications we wrote and ported for evaluation. It provides a rich environment for programming in-memory data structures that can be shared and persisted by presenting data access as memory access within a (very) large global address space in which references to *any other piece of data* are efficiently encoded. Traditional operating system abstractions reflect the hardware they are designed for, and Twizzler is no different. Twizzler's abstractions for data access center around two core concepts: **remove the kernel from the data path**, and **enable the construction of in-memory data that is free from ephemeral context**. We will see how these core concepts manifest, both in how they enable new ways of building applications and in how they improve performance when accessing data.

THE CLAIMS

This dissertation, in addition to investigating the aforementioned principal hypothesis, makes the following claims:

1. Retrofitting existing interfaces is insufficient. Instead, the correct approach to reimagining programming in a world of changing memories is to rebuild the operating system from the ground up. Similarly, backwards compatibility, while important, is not the primary goal of a reimagined system. Applications that want to adapt to new hardware trends should get first-class support.

2. A global address space of all data is a viable design for long-lived, in-memory data structures, and access to that address space can be done efficiently with little kernel involvement[12].

[12] See Chapter 5.

3. The implementation of references within the global address space matters beyond simple performance tradeoffs. We can encode references within the address space to not only be *invariant—i.e.* not based on any local context—but we can also encode them efficiently, despite the address space being large, with less space overhead than alternative approaches[13].

[13] See Chapter 6.

We will examine these claims in more detail throughout the following chapters, keeping in mind the goal—that by providing in-memory data structures that don't require kernel intervention in the data access path we can center programming around data instead of actors. Showing a viable approach to low-coordination global data naming and accessing is the primary piece of the puzzle. But after we place this puzzle piece, there is still much work to do—the operating system must provide basic services, convenience libraries, interfaces for ensuring safety in failure-atomicity and type correctness for persistent data, and (last but *certainly* not least) security[14]. These aspects are no less important than the enumerated claims, however we will approach them in such a way that ties them back to be fundamentally derived from the core concepts of Twizzler.

[14] See Chapters 7 and 8.

THE SIGNPOSTING

"Being lost is not a matter of knowing where you are. It's a matter of knowing where you aren't."

— *The Phantom Tollbooth*, Norton Juster

Chapter 2 discusses, in detail, the hardware trends we are considering and the implications they hold, followed by a discussion of how those trends will inform our operating system design work.

Chapter 3 covers the software demands of hardware and interfaces, primarily focused on the overheads that software has to deal with to get around "the context problem". We will cover serialization[15] for persistence and distribution as well as patterns of RPC.

[15] *"Boo"ing sounds.*

Chapter 4 combines the insights from the previous two chapters and discusses data-centric operating system design, introduces Twizzler, and provides an overview of relevant operating system, persistent memory, and systems research.

Following Chapter 4, we begin to focus more specifically on the design and implementation choices we made for Twizzler, studying them in case studies, performance analysis, and modeling. **Chapter 5** covers the design of the global address space in Twizzler, the model of data objects Twizzler uses, and models the collision possibility of object IDs within the space. Finally, it discusses how the global address space is realized and accessed on existing hardware.

Chapter 6 discusses how we encode references within the global address space, introduces the *foreign object table*, our solution to naming data in an invariant fashion, and performs case-studies on using invariant references to build real data structures that serve as a backend to a ported version of SQLite. These software are then evaluated for performance and compared to other solutions to persistent memory programming.

Chapter 7 enumerates several core aspects of Twizzler as an operating system, such as object services, program instancing, threading, security, and UNIX compatibility.

Chapter 8 goes over higher level programming concepts, such as failure-atomicity, type safety, memory safety, crash consistency, and new patterns enabled by Twizzler's object and invariant references model.

Chapter 9 concludes with a look back on the presented work and a look to the future for operating systems, Twizzler, and data-centric designs.

Twizzler is open-source and can be found at `twizzler.io`[16].

[16] You can tell that it's a noteworthy project because the GNU config.sub file recognizes it.

2 FAR OUT MEMORY HIERARCHIES

SYNOPSIS *The world is changing around us! This chapter introduces the hardware trends that we observe and discusses some of the details behind our expectations of how these trends will pan out. We will discuss persistence, including both SSDs and NVM, along with memory disaggregation, interconnect technologies, and device controller complexity.*

"I never am really satisfied that I understand anything; because, understand it well as I may, my comprehension can only be an infinitesimal fraction of all I want to understand about the many connections and relations which occur to me, how the matter in question was first thought of or arrived at…"

—Ada Lovelace

Operating systems provide abstractions for data access that reflect the hardware for which they are designed. For example, our current I/O interfaces reflect a structure that takes as axiom the separation between volatile and persistent data domains. Any assumptions we make about hardware, however, may become less accurate over time, resulting in an increasing impedance mismatch between the interfaces provided and the underlying hardware.

It therefore pays to examine the trends in hardware[17] and reexamine how we expose that hardware to applications. The work presented herein starts with the assumption that hardware trends are, at minimum, worth examining to see how we may wish to update our interfaces. We argue further that the trends and opportunities we are presented with demand a full scale revolution in how programming models are designed, and in the design of operating systems that wish to support new models. Let's first explore some hardware trends, and then, in the next chapter, put those trends in the context of the software that we wish to better support.

One of the most basic goals is to build up some abstractions in order to make the system convenient and easy to use. Abstractions are fundamental to everything we do in computer science. Abstraction makes it possible to write a large program by dividing it into small and understandable pieces [...] Abstraction is so fundamental that sometimes we forget its importance.

—Operating Systems: Three Easy Pieces [3]

2.1 MEMORY AND COMPUTE

Memory and storage are getting closer *and* further away[18]. Closeness of memory to CPU in latency space has long been a vital part of ensuring speedy computation. However, in the constant battle between latency and throughput (and complexity and density), latency often loses, and the latencies of DRAM have not seen dramatic reductions for some time. Meanwhile, access to *persistent* memory is getting faster in latency *and* throughput. Not only are SSDs improving, but so too are the protocols they use and the link speeds between them and main memory [77, 128].

[17] And software, as we will see in the next chapter.

[18] Bear with me, now!

Recently, we have also seen commercially available byte-addressable Non-volatile Memory (NVM) in a DIMM form-factor. This technology, Intel Optane[19] (using 3D-Xpoint technology), provides a persistent memory with low latency—only 1.5–$8\times$ the latency of DRAM in most

[19] We will discuss the outcome of Intel's "attempt" to market this technology later.

cases [61]—that is accessible from the CPU via normal load/store instructions. Direct, low latency access to NVM opens the door to building systems around persistent data structures and avoiding serialization[20].

[20] Indeed, this is what we will do.

We can view the emergence of NVM as an extreme example of the general trend of persistence getting closer to compute. As a result, the relative cost of indirection and kernel involvement in the data path is increasing. While before the cost of a system call was significantly less than that of the actual device activity, now we are seeing the cost of device access shrinking to a similar magnitude of the cost of a system call, or in the case of NVM, even less.

2.1.1 *Distribution*

However, to stop here would be a mistake. Another significant trend is *disaggregation*, in which memory is placed across machines and programs are written expecting various levels of access to this distributed memory. This is, admittedly, as much about software as it is about hardware, however there are underlying hardware mechanisms that are driving the push toward decoupling memory and compute. On a network level, renewed interest in Distributed Shared Memory (DSM) is sourced from a combination of several factors:

1. Network speeds are continuing to increase. With technologies like 100 Gigabit NICs and 10 Tb/s switches, previous limitations on distributed memory that arise from network performance reduce or go away entirely.

2. Network hardware complexity is increasing. Not only can those switches sustain 10 Tb/s of throughput, but they can also be programmable, allowing the network to have much more participation in protocols. Furthermore, we are seeing protocols like RDMA enabling remote memory access via the network.

It isn't just traditional networking in which memory semantics are leaking out past a single traditional node. Interconnect technologies like CXL have the potential to upend the traditional view of "CPU attached to memory with peripheral devices" as the fundamental concept of a node. These trends mirror the persistence trends above—things that were traditionally far away (other nodes' memories) are getting closer in a logical space, and things that were close (local DRAM and storage) can be more efficiently shared.

Finally, improved fabrication techniques and shrinking feature sizes means hardware controllers are increasing in complexity, offering more autonomy from the CPU, off-CPU processing capabilities, and better parallelism. Hardware interfaces reflect the controller complexity expected of devices; for example, AHCI controllers improved request queuing over ATA, and DMA allows hardware to copy data directly to and from memory independently from the CPU. NVMe expands the responsibility of controllers again, adding deeper and parallel command queues, requiring devices to multiplex requests themselves and allowing them to exploit the parallelism of access available in SSDs. We expect these trends to continue, resulting in more programmable hardware devices that are able to act on shared, global memory with more autonomy.

2.1.2 *Memory Density*

Another point to make is one of increasing memory sizes, largely stemming from increased density of memory technology. Not only can we see this in DRAM, with well-known systems taking advantage of increased DRAM sizes [94], one of the major selling points of Intel's Optane NVM was a significant increased density and lower cost per byte. Having larger memories means a few things:

1. Some nodes may have wasted memory as workloads come and go. This ties in with the above points about disaggregated memory—sharing unused memory is more cost effective, since unused RAM is wasted RAM.

2. Memory can have an induced-demand effect. Similar to how building additional lanes on highways often just increases traffic [17], making more memory available can cause additional memory traffic since working sets can be larger.

3. In-memory sharing can increase. With significantly larger memories, applications can take advantage by not writing intermediate results to storage, and instead can write to shared memory.

2.1.3 *The Heterogeneity of Memory*

The advent of NVM in a DIMM form factor has given us further motivation to consider the effects of a heterogeneous memory environment. When all memory is a single "kind" (*e.g.* DRAM), it matters little from where in the physical address space memory is allocated, since devices (including the CPU) can only access one kind of memory between them[21]. However, if our physical memory is comprised of multiple kinds of memory, we must be more careful with our allocation. Programs may request allocations of memory for their data based on different policies of what kind of memory they want—volatile versus persistent, high-bandwidth versus DRAM, to name a few.

[21] Yes, older devices are limited in address space, but this is not important, and I do not wish to get stuck in the past.

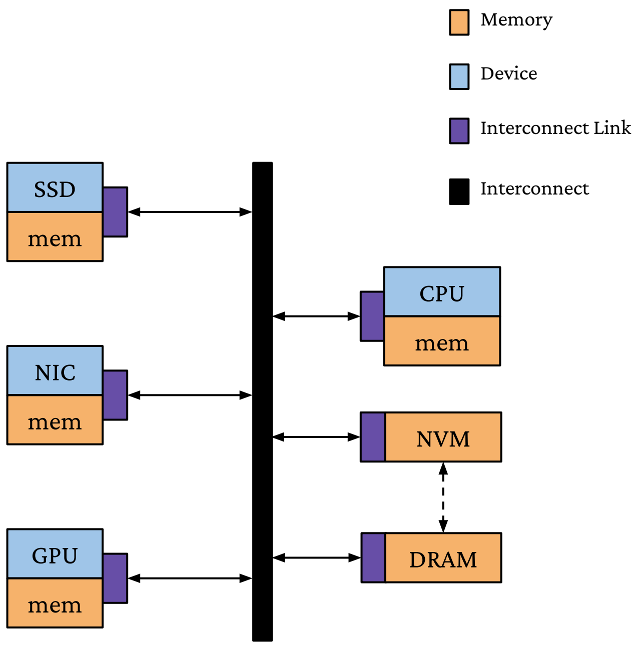

FIGURE 2.1

Our expected system architecture. Solid black lines show data paths. Dashed lines and boxes show potential dedicated paths.

But it's not just allocation policy that is affected. Having different kinds of memory means that large-scale movement of data between regions of physical memory is now semantically meaningful and may be triggered as a result of policy, evictions, *etc*. Interacting devices, therefore, must coordinate on a shared mapping of logical data to physical data, as the physical data might move between kinds of memory. One could easily argue that this heterogeneity is not actually new—with NUMA, we see many of the same problems. I agree! However, the correct solution is to design around a fundamentally heterogeneous memory system and build the concepts of data movement across physical memory into the core abstractions. Such a model is not only useful for the advent of NVM on the memory bus, but it also covers future additions of different kinds of memory as well as subsuming the NUMA model.

2.2 SOME KIND OF MODEL

These changes are coupled with a shift in focus for how data moves throughout the system. Traditionally, data is considered to move into main memory for computation by the CPU, followed by storage to persistent memory (through a disk or SSD controller) or moved out onto the network (through a network controller). Figure 2.1 shows a different model, where data is able to more fluidly move through the system, or even needs less movement. With more complex controllers and off-CPU processing, we expect non-traditional data paths to become more common. Say, for example, a packet arrives at the NIC containing compressed data for GPU processing. A traditional system would move the compressed data into main memory, decompress it using the CPU, and move it into GPU memory for processing. Instead, we could see a dedicated compression chip in the NIC whose job it is to decompress incoming data. That data could then be moved directly between the NIC's buffers and the GPU memory, without involving the CPU and main memory at all. This both reduces copies and frees these resources for other uses.

Note the rather generic model in Figure 2.1. We are intentionally not ascribing specific properties to the components protrayed within. Attemping to build generic system abstractions atop an abstracted model while not losing too much in the abstraction is a fundamental essence of operating system design. Here, we are trying to capture the basic ideas of the trends we discussed—individual, higher powered devices, accessing shared memory pools and possibly each others' memory pools, with possible NVM and fast interconnects that allow addressing control and isolation[22]. Concretely, we can see this model mapping well to, for example, CXL, and even current PCIe systems[23]. As the trends we observe continue, the inherent concurrency enabled by allowing devices to operate more independently with more ability to share memory without waiting for permission from the CPU and the reduced lengths of code paths in processing data points strongly to a model like the one we describe here.

[22] Cache coherence is an issue that we are not explicitly designing for or against. It remains to be seen how cache coherence between these devices will play out. Nonetheless, the issues of coherence and failure-atomicity are not lost on us, and we will discuss them in Chapter 8.

[23] If we squint our eyes and have perhaps undue faith in the IOMMU.

2.3 SO, WHAT DOES THIS ALL MEAN?

The trends above imply several basic requirements for a set of operating system interfaces:

REMOVE THE KERNEL FROM THE DATA ACCESS PATH The cost of system calls is too large relative to device access to justify their use in the data path. We must provide lightweight, direct, memory-style access for programs to operate on data.

ASSUME DATA LIFETIME IS DISCONNECTED FROM EPHEMERA Memory can now be pooled in units outside the purview of the CPU. This can happen either in time—data can be persisted, where the CPU loses control over that data when power is cycled—or in space—where data can traverse a network, or be accessed by multiple devices on an interconnect.

While these forms of temporal and spacial sharing have always existed, they have either been relegated to "experts" in cordoned off low-level areas of the system (drivers and DMA) or have been explicitly designed around by, for example, building OS abstractions around an assumed bifurcation of data into "volatile" and "persistent" domains, and asking programmers to explicitly move data between them[24].

[24] We will explore this more fully in the next chapter.

However, when the previously outcast domains of persistence and network become closer to compute as we saw above, either via faster access or pushing compute power further out, we must **design for in-memory data structures**. Long-lived data structures can directly reference persistent data, so references must have the same lifetime as the data they point to. Virtual memory mappings are, by contrast, ephemeral, and so cannot effectively name persistent or distributed data. But perhaps more importantly, in-memory data structures are *what we compute on*, and it's *computation* that matters most. If we can reduce or even remove extraneous processing paths that waste time, we should take that opportunity.

TOWARDS A DATA-CENTRIC OPERATING SYSTEM We call an operating system that meets both of the above requirements *data-centric*, as opposed to current OSes, which are *process-centric*. Facilitating operations by applications on in-memory data structures is the primary function of a data-centric OS, which tries to avoid interposing on such operations, preferring instead to intervene only when necessary to ensure properties such as security and isolation. To meet both of these requirements a data-centric OS must provide effective abstractions for identifying data independent of data location, constructing persistent and distributed data relationships that do not depend on ephemeral context, and facilitating sharing and protection.

WHY NOW? It is natural at this point to wonder, "why now? The memory hierarchy has *always* been changing." This is true, but there are a few unique elements to the recent changes we are seeing. First, the placement of NVM directly on the memory bus is a fundamental dramatic shift in how we view persistence and opens the door to *true* single-level stores. Second, the increasing distribution of both memory and CPU requires that we rethink how computation is delegated throughout the system and how those separate memories and devices can organize, access, and reference data. But, more fundamentally, this thesis argues that our data-centric model was *always* the correct model. We just have an opportunity to realize it, and modern hardware has the ability to make it efficient and scalable. The memory hierarchy has always been changing, but our interfaces have not kept pace. It is time for a revolution, designed for new hardware and, as we'll see in the next chapter, informed by software.

3 THE DEMANDS OF SOFTWARE

“So many things are possible as long as you don’t know they’re impossible.”

—The Phantom Tollbooth, Norton Juster

SYNOPSIS *This chapter will focus on software, the needs of software we must address, and the perspective of systems programming, putting the hardware trends we spoke of last chapter into context. We will discuss the problem of ephemeral context and discuss how persistence and RPC have shared characteristics and concerns.*

At their core, applications perform computation on in-memory data. Yet much application complexity is tied up in the infrastructure surrounding that computation, for example, in managing persistent state and in performing serialization. Additional work done by the program (or by the operating system on behalf of the program) that is merely in service of setting up the computation and not the computation itself is considered *overhead*. Our goal is to reduce that overhead in two ways: improving performance and reducing complexity. To do this, we need to understand where overhead comes from, which elements are fundamental, and which are not.

One goal in designing and implementing an operating system is to provide high performance; another way to say this is our goal is to minimize the overheads of the OS. Virtualization and making the system easy to use are well worth it, but not at any cost.

—Operating Systems: Three Easy Pieces [3]

3.1 THE OLD WAYS AND “SYSTEMS PROGRAMMING”

Current operating system interfaces are a poor fit for the trends and hardware requirements we discussed last chapter. File `read` and `write` interfaces, originally designed for sequential media and later expanded for block-based media, require significant kernel involvement and often serialization, violating both the requirement to reduce kernel involvement, and the need to reduce the length of code paths around accessing data. Figure 3.1 shows a fairly common data path for an application that operates on (and perhaps mutates) some persistent data. First the data is loaded explicitly into memory, either in a streaming fashion or as a whole file, followed by the application manually deserializing it into an in-memory form. Once this is done, the program may commence its actual purpose—to perform computation—after which the results are serialized and placed back onto disk with an explicit store operation.

Let’s consider the case from Chapter 2 of faster, closer persistence. In the past, the overhead in manually loading and unloading data and in transforming it to and from an on-disk form was acceptable, since the cost to access the disk was high, and those explicit load and store disk operations were going to be issued anyway. But when these operations

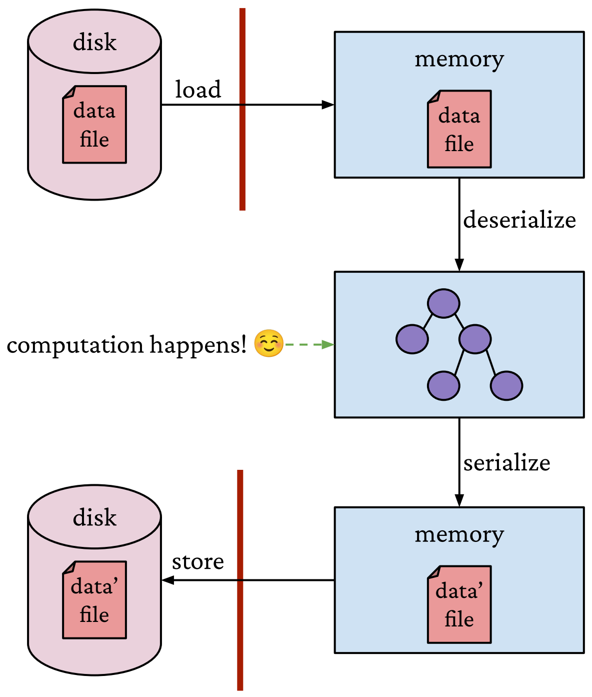

FIGURE 3.1
The "standard cycle" of systems programming. We start with some persisted data, explicitly load it from disk into memory, deserialize it, compute on it, reserialize it, and explicitly store it back to disk. Note that the vast majority of this diagram depicts stuff other than the actual computation. The thick red lines depict persistence boundaries, across which data exists in different contexts and assumed lifetimes.

overshadow device access time, the overhead quickly becomes unacceptable. As devices become faster and as applications process more and more data, we find that more and more of our processing time is spent here. Applications can spend as much as 70% of the processing time [30] deserializing and loading data into main memory at request time. Reducing this overhead would save both in program time but also in *programmer* time. Simplifying applications by removing serialization helps programmers by removing the need to maintain multiple data formats and the transformation code to move data between those formats.

3.1.1 *The Context Problem*

If we look closely, the cycle in Figure 3.1 can represent more than just processing files. It works equally well for receiving data from a network and sending back a response (microservices and servers in general), or for applications in a traditional `pipeline`, *etc*[25]. But why do we often spend so much time and effort with serialization and explicit I/O? Because the POSIX abstractions fundamentally bifurcate our programming model into two separate *contexts*:

1. **Local Context** contains data that is limited to a single domain, for example in-memory data accessed via virtual addresses by a single

[25] Those who sit high atop their Unix throne claim that text is a universal communication language. Yet anyone who has actually had the misfortune of parsing the output of even a moderatly complex Unix tool can realize that systems programming here involves just as much serialization and processing work as anywhere else—or else involves doing computation directly on C strings, a fate I would not wish upon my worst enemy.

process, or the domain of file descriptors. Data in a local context is typically operated on "directly" since it's usually in memory. Most importantly, though, the data isn't *shared* with other domains, and it is ephemeral—its lifetime is tied to the context it is in.

2. **Global Context** contains data that is shared across all the domains. Typically this covers data stored on disk or other stable storage. Data in a global context is usually stored in some serialized format and can be accessed by multiple domains simultaneously. The lifetime of data herein is, in the limit, infinite[26].

[26] Even in the global context, we traditionally separate local storage and network storage. That separation can be eroded depending on the programming model, however, as we will discuss later.

These separate contexts give rise to the need to transform data between them and to be explicit about marshalling data across the boundary. The primary reason for the need for transformation has to do with *data references* and *naming data*. For example, when applications operate on in-memory data, references take the form of virtual addresses—pointers refer to specific locations within the virtual address space. The virtual address space is ephemeral—it's tied to a specific process. But *persistent* data is *not* ephemeral! By definition it is accessed by multiple processes, either over time (different invocations of the same process), or across space (shared between multiple processes), or both. In either of these cases, virtual addresses cannot be used, since those addresses may not agree across different contexts. So when sharing memory, any references to local context data must be transformed into something that can be interpreted within the global context.

More generally, we often lock data access behind context that is tied to a shorter-lifetime actor, leading to unnecessary indirections and extraneous work that the program must perform. This is illustrated in how data relationships are typically handled, either:

1. **Explicit, context-sensitive.** As we saw with virtual address pointers, references between data are encoded explicitly but rely on context provided by the ephemeral process abstraction. These references cannot be reliably shared between applications and across nodes which do not have the context necessary to interpret them.

2. **Implicit.** Many data relationships are implicit in applications. Although there is a relationship between the UNIX files `/etc/shadow` and `/etc/passwd`, it is not explicit. This limits interoperability between programs, prevents relationship discovery, and results in a brittle environment if these files are moved or replaced.

3. **Explicit, via mediators.** Relationships can be encoded explicitly without ephemera if we use a global name resolution service. For example, an HTML file can refer to a style sheet by name. However, this presupposes the existence (and availability) of a global

mediator and restricts programs from agreeing on the identity of data behind a reference without complex inter-networking and expensive global coordination.

In our view, the complexity of these mechanisms is a symptom of a more fundamental problem: access to long-lived data is unnecessarily mediated via ephemeral actors. We advocate a violent break from this actor-centric model of data access, in favor of a model that elevates *data* as the systems' first-class citizen. Doing so will require *explicit context-free* references via globally unique identifiers (GUIDs) that name data objects. These references are independent of context (*e.g.* process) that operates on them, and require that all context necessary to interpret references be stored within the data itself. Of course, we still need some understanding of context-sensitive relationships, for example late-binding of names. We do this via a "two-level" naming system in which GUIDs are *authoritative* names to which we can bind additional names for purposes like local references, changing data identification, and discovery[27].

[27] We will discuss these in more detail in Chapter 6.

Viewing the past through the lens of the present, it was *always* a mistake to entangle the context required to access and manipulate data with ephemeral actors. However, in the days of slow networks, spinning disks and small memories, it was possible to hide these complexities and inefficiencies. For one thing, disk-based I/O led to a model of sharing in which long-lived data and computable data were stored in different *kinds* of memory, in different formats (due to serialization), with different reference formats (e.g., file names vs. virtual memory pointers). The filesystem was a natural location to place the various hacks that were required to paper over the reality that interacting with long-lived data required figuring out how to name the short-lived processes that were the primary citizens of the operating system.

Needless to say, a disaggregated system model—one in which a particular data reference may ultimately point to data on a different node from the one observing the reference—exacerbates all of these problems! Two different nodes that observe the same data reference should agree on what data is being referred to, and by the same token, an individual node observing two different data references should be able to determine if they point to the same data. This asks the question of *how* to encode these explicit context-free data references while remaining efficient and avoiding global coordination[28].

[28] We will discuss and answer these questions in detail in Chapters 5 and 6.

3.2 RETROFITTING POSIX?

While I am arguing strongly that a clean-break reimagining is needed, it is worth considering what an incremental approach might look like. For example, the problem of direct access might be solved to some extent by mmap, the problem of global references could be solved by traditional pointer swizzling built atop mmap, *etc*. We could even combine this with RDMA to gain a DSM model.

Let's first consider the use of mmap to provide an interface for NVM before looking more broadly at persistence in general. The mmap system call attempts to hide storage behind a memory interface through hidden data copies. But, with NVM, these copies are wasteful, and mmap still has significant kernel involvement and the need for explicit msync calls. "Direct Access" (DAX) tries to retrofit mmap for NVM DIMMs by removing the redundant copy, but this *still* fails to address the problem of global context and in-memory data structures for persistent storage. Operating on persistent data through mmap requires the programmer to use either fixed virtual addresses, which presents an infeasible coordination problem as we scale across machines and is fundamentally unportable, or virtual addresses directly, which are ephemeral and require the context of the process that created them[29].

[29] These issues don't even touch the high kernel involvement, in-kernel coordination, and failure-atomicity hazards present when using mmap [29].

Attempting to shoehorn NVM programming atop POSIX interfaces (including mmap) results in complexity that arises from combining multiple partial solutions. Given some feature desired by an application, the NVM framework can provide an integrated solution that meshes well with the existing support for persistent data structure manipulation and access, or it can fall-back to POSIX resulting in the programmer needing to understand two different "feature namespaces" and their interactions. An example is naming, where a programmer may need to turn to the filesystem to manage names in a completely orthogonal way to how the NVM framework handles data references. For example, PMDK, an NVM programming library, relies on a filesystem for naming and initial access to persistent memory objects, resulting in different kinds of references, feature sets from filesystems being applied (like security) while others are not (data access), and the complexity of understanding how the PMDK abstractions interact with the POSIX ones. Instead, our model prefers to build legacy support atop new abstractions (as we will see in Section 7.4), and avoid falling back to legacy models for persistent data access.

Even without NVM, we have similar problems. Our goal of programming with memory semantics on in-memory data structures leads to many of the same arguments as above. Additionally, systems that layer new models atop existing interfaces often fail to facilitate effective persistent data sharing and protection. PMDK, for example, makes design choices that limit scalability, since its data objects are not self-contained and do not have a large enough ID space, resulting in the need to coordinate object IDs across machines, a problem we will explore in detail in Chapter 5. For the same reason, although single-address space OSes [19] somewhat address our requirements, they do not consider both requirements at once, nor do they provide an effective and scalable solution to long-term data references due to that same coordination complexity (as we will discuss in Section 4.4).

3.3 REMOTE PROCEDURE CALLS

One major aspect of systems programming, particularly in distributed systems, is RPC[30]. We typically use RPC as a mechanism for decomposition in distributed systems, allowing us to break design problems down into smaller, re-usable parts that hide implementation details and are more easily debugged. Decoupling components with RPCs allows them to scale independently—in principle, developers need only agree on a common interface and message format to leverage the benefits of software decoupling. Yet, in reality, RPCs enforce strict interface constraints and often trade adaptability (narrow interfaces are harder to evolve) for simplicity (narrow interfaces limit cross-component interactions), ultimately hampering the goal of scalability.

[30] Note that, if you look at Figure 3.1 and squint your eyes, it resembles not just persistence but also communication and data transfer across nodes.

The chief problem with RPCs is that they are fundamentally location- and compute-centric: RPCs force a programmer to decouple an application by explicitly separating the computational endpoint or *location* where a function is invoked from the location where the function executes. As a consequence, they are well-suited to a relatively narrow set of use cases in which function arguments, which flow from invoker to executor, and returns, which flow back, must be serialized and sent in their entirety, and hence are small, and in which reference data must be located on the executor.

Many scenarios would benefit from decoupling but are simply not feasible using existing RPC mechanisms. For example, the invoking endpoint may have abundant data but limited compute, the invoker may wish to traverse a remote data structure, or the invoker may wish to refer to data that they lack privileges to read. Rapidly growing model sizes, privacy concerns, and the proliferation of last-mile model customizations all exacerbate the issue. To mitigate the problem of location-centric RPCs, data center operators often deploy discovery services, load balancers, or other forms of middleware [40, 59, 72, 111, 120]. These extra indirection layers make the execution endpoint abstract, but at the cost of increased latency and added system complexity. Moreover, we argue that such systems do not address the fundamental problem, which is that *we need a more general mechanism for module composition in distributed systems*.

Motivating Example

To illustrate the poor fit of RPC as a decoupling mechanism for some classes of applications, consider the problem of distributed inference for

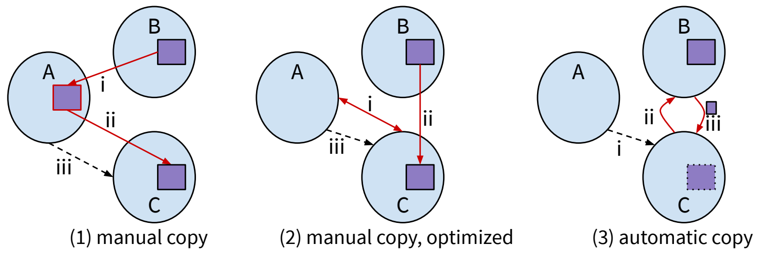

FIGURE 3.2
Rendezvous of data and compute. Solid red arrows are additional infrastructure-level tasks that are not fundamental to the requested computation.

edge devices. Here, sensors in mobile devices with modest processing and storage resources (*e.g.* mobile phones) are the source of observations used both for training and inference. Recent work has focused on decoupling and distributing machine learning training across edge and cloud resources to minimize client-perceived latency, provide privacy guarantees, and maximize server-side throughput [68, 113].

In this example, we focus on the inference problem that arises in response to device input. Ideally, small models trained in the cloud (via a methodology such as federated learning) are periodically shipped in their entirety to edge devices, which perform local inference. Several trends are upsetting this model. The first is the aggressive growth—roughly 10×/year—of models, in particular language models. In 2018, the largest machine translation models at Google were 8.3 billion parameters [112]; a mere two years later, the largest models exceeded 800 billion parameters! Inference on sparse giant models which far exceed device resources must be performed server side, where model serving presents a substantial throughput bottleneck. This is further compounded by last-mile model customization for end users, in which inference tasks for different devices *must* be performed on slightly different models. As much as 70% of the processing time [30] for these model-serving applications is spent deserializing and loading the sparse personalized models into main memory at request time. Finally, users prefer local models remain local due to confidentiality concerns.

Consider a concrete example (Figure 3.2) that is bedeviled by all of these complexities at once. A mobile device, Alice, in possession of a locally-trained model and an activation, wishes to perform a classification task that requires a partition of a sparse global model, located on cloud resource Bob. Further, imagine that Alice cannot perform the inference locally, either because the global model fragment is too large or because she has inadequate local compute. Finally, imagine that Bob is overloaded, while a separate cloud resource, Carol, is mostly idle.

An ideal solution will minimize the latency Alice perceives and maximize the throughput offered by both Bob and Carol, all while satisfying capacity constraints of each. It is easy to see that while this application requires decoupling, RPC is the wrong abstraction in terms of performance, expressivity, and flexibility. Data movement—whether from storage to DRAM on Bob or from Bob to Carol—requires costly serialization. If moving the data to Carol and performing inference there is the optimal solution, the application logic on Alice must orchestrate this infrastructure-level concern, either by pushing the data through Alice (Figure 3.2 (1), a naïve solution) or having the RPC executed on Carol address Bob directly and pull (Figure 3.2 (2)). Both (1) and (2) require additional logic on Alice to work—the programmer had to perform the infrastructure level task of data movement. Fundamentally, these issues arise because the system's core abstraction is location-based—the programmer is forced to manually orchestrate machines or resort to costly copying[31]. Further, heterogeneity among end devices makes the "hard-coded" data movement strategy brittle: a subsequent classification request from client device Dave will be forced to run inference on the server side even if it is equipped with the resources to do the work locally.

[31] Additionally, there are issues of security and coordinating topology changes.

Figure 3.2 (3) is more ideal. The first step is for the computation to move to Carol instead of first moving data in preparation. This may seem like a minor point, but it is not—by specifying *up-front* the computation we want to perform, we open the door for lower-level optimization to examine our requests before we go around manually moving data. In fact, the programmer may not be directly asking Carol to perform the computation; instead the placement decision would be made by the system. Once the code starts executing, we can then move data on demand instead of having to move the entire object. Note that this matches the same requirements we discussed above with respect to context—data references must be *context-free* and global. With the ability to name data in a global context, we not only reduce the need for serialization, but we also gain the ability to pass by *reference* instead of by value.

PATTERNS OF RPC. While RPC is a poor fit here, there are applications for which RPC provides adequate decoupling. RPC shines in situations where decoupling in the application meshes well with having little data movement, where an RPC endpoint either fronts large data, large compute relative to the invoker, or some combination, with *small* arguments and return *values*. But call-by-small-value is a significant constraint, and there are many classes of applications that do not fit. We cannot paper over this problem, either. Because RPC is disconnected from a global notion of data identity, we either *cannot* do call-by-reference (because references must cross machine boundaries) or we must shoehorn this functionality into the application logic and the RPC's APIs, resulting in brittle, repetitive, complex code to deal with the coordination, caching, and prefetching that comes from distributed data and global references when data moves.

The "good" use cases of RPC are ones in which code and data colocation has already been preordained by initial decoupling (after which it is rigid) and data transfer is minimal—often manifesting as something like a fronted key-value store service. This restricts code mobility (as it accounts for no change to decoupling later), and requires a myriad of RPC calls to implement all the ways a programmer might wish to view data (one need only look at the many S3 APIs available as an example). If we limit ourselves to traditional RPC, any situation (such as the one discussed above) that does not fit this pattern either results in expensive data movement and complex application logic, or it must be dismissed altogether and the application redesigned. By allowing applications to pass data references instead of just values, and by making data references a first-class abstraction in the OS and the network, applications become much simpler to express efficiently, even for what would be considered pathological cases for RPC.

a counter to a defense of serialization A common response at this point is to point out that serialization serves several other purposes, such as ensuring portability between environments that use different definitions of (*e.g.*) `int` in C, or different machine architectures that differ in endianess. I find these defenses weak. Firstly, for worrying about different ABI of types, this is already a problem when sharing memory, and is a solved one. Many languages offer support for fixed-width values and have explicit FFI functionality[32]. As for endianess, much of the modern world uses little endian, and places that use big endian (networks) don't really care about the data they are passing, they just worry about protocol headers. In any case, both of these issues are problems that we can (and regularly do) solve at the language level.

[32] And continuing in the modern day with C's type system is a complete misstep anyway.

And furthermore, there is nothing fundamentally tying these elements to the idea of global context and references, so even if they weren't concerns at all, we would still often need to serialize data as it moves across contexts. Our approach is to solve the fundamental context problem that motivates serialization now, and solve the smaller problems that got mixed into serialization at a higher level.

3.4 so, what does this all mean (part 2)?

In the early days, building computer systems was easy. Why, you ask? Because users didn't expect much. It is those darned users with their expectations of "ease of use", "high performance", "reliability", etc., that really have led to all these headaches. Next time you meet one of those computer users, thank them for all the problems they have caused.

—Operating Systems: Three Easy Pieces [3]

Software is growing in complexity, software is becoming more and more distributed with higher and higher demand for concurrency as it is asked to process more and more data. As we saw both with accessing persistent data and in RPC, serialization takes a huge amount of processor time and a large chunk of an application's complexity budget. Relying on kernel crossings to perform I/O is costly, particularly if we reduce or remove the need for serialization. Having to resort to call-by-small-value when trying to effect computation remotely severely limits the expressability of distributed applications, and can have a negative impact on decoupling and modularity—the very thing that RPC is supposed to solve.

If we are to provide a solution by way of a new programming model and OS abstractions, it must be one that both solves the problems that software has and provides abstractions that best take advantage of the new hardware. Mere incremental change will not get us there. Fortunately, what we are seeing is a confluence of software demands and hardware trends. The hardware is becoming better equipped to support high concurrency, distributed applications that can share memory within a larger context. To take best advantage of hardware, we must reduce kernel involvement and move towards a global address space of in-memory data—which is exactly the solution that cuts down the overhead of serialization and system calls to zero. The remainder of this thesis will discuss how we can use this global address space, with first-class support for references, to solve not only the problems of persistence and the systems programming cycle, but also that of computation mobility, factoring out complexity, and providing simpler, higher performance applications. But to do this, we will need an operating system built for that purpose.

Part II

A DATA-CENTRIC OS

The most dangerous phrase in the English language is "we have always done it this way".

—Grace Hopper

4 THE DATA-CENTRIC APPROACH

It's the wanting to know that makes us matter.

—*Arcadia*, Tom Stoppard

SYNOPSIS *The last few chapters discussed the hardware trends and the modern needs of software. This chapter will take those insights and define a design space within which we will build a data-centric OS. We will then conclude with a look back towards related, prior work.*

4.1 AN OPPORTUNITY ARISES

The confluence of hardware trends (Chapter 2) and software needs (Chapter 3) demands a fundamental shift in the way we program computers. Our two-tier memory hierarchy does not adequately reflect what hardware is trending to provide nor does it provide adequate abstractions for applications to operate on data with low overhead. Meanwhile, the increasing heterogeneity of our computing environments—different kinds of memory at different distances, specialized computing devices, near-storage compute, and remote computation—force us into a corner when trying to use current operating system abstractions to model what is, in essence, a global environment of computation and data. It is vital that operating systems evolve to provide new abstractions—they must make the best possible use of hardware trends to provide a programming model that aligns with software's demands for low-overhead computation. Mere incremental change is insufficient for enabling dramatic improvements in performance and complexity.

Presented with these trends, we are building a *data-centric* operating system, one that eschews the traditional process-centric (or actor-centric) model of operating system design in favor of making *data* the primary citizen. In process-centric models, the application is the first-class citizen, operating on data in isolation, performing explicit persistence and network activity, and relying heavily on serialization. Instead, as we will see, a focus on data in a global space makes it possible to enable better sharing (both between applications and devices on one machine and between nodes), lower overhead computation, and simpler applications.

Clearly, for computer systems to be interesting, both input and output are required.

—Operating Systems: Three Easy Pieces [3]

Fundamentally, data is and has always been the focus of programming. In the past, our abstractions have failed to adequately capture this simple fact. But now, as software looks towards more modern programming models and the friction between hardware and current models grows, we have an opportunity to reimagine how we structure programs and data. It is a convenient time to make a break, when the impedance is low—software will need to change to take advantage of new hardware, and we should support those applications with better fitting models.

4.2 A DESIGN SPACE

The idea of a data-centric operating system in which in-memory data is the primary citizen still leaves a large space for how we might actually design such a system. Indeed, the remainder of this dissertation will be focused mostly on our design and implementation choices, the rationale for those choices, and empirical evaluation. However, it is still worth taking a minute to discuss the design space we are in and nail down what it means to design a data-centric OS that meets the requirements set out in the previous chapters.

LOW KERNEL INVOLVEMENT As we saw earlier, the kernel imposes a heavy overhead on applications that wish to access data outside their local context, even in the case of interfaces like `mmap`, particularly as device latency continues to drop. A data-centric OS should, therefore, avoid requiring significant kernel intervention, instead preferring to allow applications to access data directly as in-memory data structures[33]. The reduction in kernel involvement and responsibility has some wide-reaching effects, which we call "the death of the process".

[33] See below.

Processes as a first class OS abstraction are, like virtual addresses, unnecessary; a traditional process couples threads of control to a virtual address space, a security role, and kernel state. However, with the kernel removed from persistent data access, much of that kernel state[34] is unnecessary, leading to a decoupling of mechanisms: nothing fundamentally connects a virtual address space (a piece of ephemeral context used to access data) and a security context (*what* data threads may access). Instead, a data-centric OS can keep the good parts of a process but *separate* virtual address translation and security roles, allowing threads to select one of each as needed.

[34] *E.g.* file descriptors, file abstractions, *etc.*

The process abstraction is just one example. Persistent data access and sharing plays a key role in OS abstraction design, and we need to avoid complexity arising from combining old and new interfaces. Hence, we need to consider the wide-reaching effects of changing the persistence model on *all* aspects of the system, not just I/O interfaces. These hardware trends give us an opportunity to design an OS around the requirements of the target programming model instead of trying to mold support libraries around existing interfaces. While it is important that we provide support for legacy applications, it is these applications that should be relegated to support libraries; new applications built for the programming model should get first-class OS support.

IN-MEMORY DATA STRUCTURES AND GLOBAL ADDRESSING The need to place in-memory data structures front and center comes from both the desire to reduce serialization overhead (Chapter 3) but also to allow better sharing and direct use of distributed or persistent memory (Chapter 2). As we discussed, the virtual address space does *not* work for this purpose[35] as it is ephemeral. Furthermore, sharing data with not just other applications but other nodes and devices demands a large and global address space of data to avoid the context problem. But we do still need a mechanism to allow the organization of data at a large scale, the coordination of that address space, and a method for naming data within a global address space that is invariant of local context. The solution to global addressing and data references makes up the bulk of the work presented here, but the result is a system that can share data with a 100% byte-level copy across nodes and between processes where the pointers *all mean the same thing*.

[35] We will discuss single address space operating systems later in this chapter.

POINTERS AS A COMMON LANGUAGE Data references are a fundamental part of encoding data relationships[36]. When we consider the implication of encoding data references within a truly global address space, we find that we can treat references as a *common language*. If all nodes, processes, and devices agree not only on names for data but also *how* to name data, they can more easily share data and interoperate. In fact, levels of the infrastructure that previously had no view into data relationships can now gain additional insight when performing tasks.

[36] See Chapter 3.

This addresses the issues with RPC that we discussed in Chapter 3. We can combine the code mobility of RPC with the flexibility offered by DSM-like models in a global address space with invariant references as a first-class abstraction. By imbuing data with fundamental identity and pushing an understanding of data references into the OS and the network, we can leverage aspects of content-based networks to reduce the coordination typically required in a shared, distributed address space. The programmer is then free to express their computation through references to code to run on some *references* to data, instead of needing to serialize and copy *values* for arguments. Today, developers are often forced to implement functionality such as caching, prefetching, and manual data movement in preparation for some operation. With data references as a common language between the OS, the network, and applications, we can move this infrastructure-level functionality out of the application and back *into the infrastructure* where it belongs.

PERSISTENCE IS SHARING In Chapter 2 we discussed how persistence is getting closer to compute in multiple ways. A natural approach would be to focus on *orthogonal persistence*, in which applications write data and it is automatically persisted. However, mere orthogonal persistence misses out on a more fundamental relationship with distribution of memory and applications. Our view is that persistence and distribution are two sides of the same coin—we saw previously that both are plagued by the same problems of overhead, and both domains are seeing dramatic improvements in performance. In a data-centric OS, persistence *is* sharing, and the problems of persistence[37] can be solved in terms of atomicity and transactions[38].

[37] Including, but not limited to, failure-atomicity and durability.

[38] As we will discuss in Section 8.3.

4.3 TWIZZLER: A POINT IN THE DESIGN SPACE

The consequences of meeting the requirements of these hardware and software trends define a bounded design space for data-centric operating systems that we layed out above. We have chosen a point in that space and built Twizzler, our approach to providing applications with efficient and effective access to new, “far out” memory hierarchies. The core philosophy of Twizzler is that **any context necessary to interpret some data must be stored with that data**. In the following chapters we will discuss our design choices, including our global address space, invariant pointers, and Twizzler OS services and libraries, and see how this philosophy, when combined with the ideas behind a data-centric operating system, enable us to build an efficient implementation of a modern approach to programming model design.

Twizzler is a stand-alone kernel and userspace runtime that provides execution support for programs. It provides, as first-class abstractions, a notion of threads, address spaces, persistent objects, and security contexts. A program typically executes as a number of threads in a single address space, providing backwards compatibility with existing programming models, into which persistent objects are mapped on-demand. Instead of providing a process abstraction, Twizzler provides *views* (Sections 5.3 and 7.2.1) of the object space, which formalizes the notion of ephemeral context within our model by allowing programs to map objects for access, and *security contexts* (Section 7.3) which define a thread’s access rights to objects in the system. Twizzler provides invariant pointers (Chapter 6) in a low-coordination global address space (Chapter 5) for programs, as well as primitives to ensure crash-consistency (Section 8.3). The thread abstraction is similar to modern operating systems; the kernel provides scheduling, synchronization, and management primitives. Figure 4.1 shows an overview of the system organization and how different parts of the system operate on data objects.

Twizzler’s kernel acts much like an Exokernel [44, 67], providing sufficient services for a userspace library OS, called *`libtwz`*, to provide an execution environment for applications. The primary job of `libtwz` is to manage mappings of persistent objects into the address space (Section 5.3) and deal with invariant pointers. Twizzler also exposes a standard library

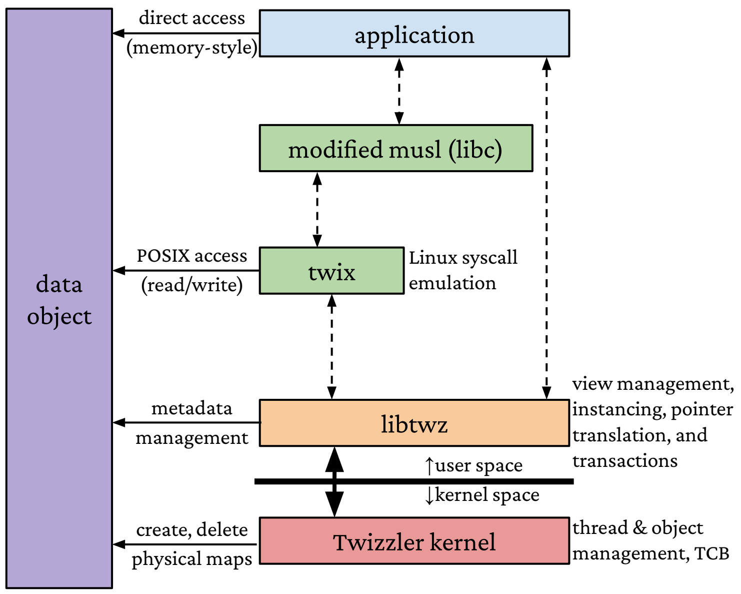

FIGURE 4.1

Twizzler system overview. Applications link to `musl` (a C library), `twix` (our Linux syscall emulation library), and `libtwz` (our standard library). Through `musl`, they may act on in-memory data with POSIX interfaces, though we expect Twizzler applications to operate directly on in-memory data with the supplied Twizzler abstractions.

that provides higher level interfaces beyond raw access to memory. For example, software that better fits message-passing semantics can use library routines that implement message-passing atop shared memory. Twizzler's standard library provides additional higher level interfaces, including streams, logging, event notification, and many others. Applications use these to easily build composable tools and pipelines for operating on in-memory data structures without the performance loss and complexity of explicit I/O.

4.4 A HISTORICAL LOOK

Twizzler's design is shaped by fundamental OS research [19, 27, 42–44, 67, 73], which, while approaching similar topics as we described previously, often did not consider all design requirements we discussed simultaneously, nor did they have a view of hardware trends that we do today, resulting in an incomplete picture. Recent research on building in-memory persistent data structures [24, 35, 39, 58, 80, 122], often focuses on building data structures that provide failure atomicity and consistency for only persistence. In contrast, we explore how hardware trends affect programming models and OS abstractions on the whole. We draw from recent work on providing OS support for NVM systems [16], as these align well with in-memory sharing data structures, and work providing recommendations for NVM systems [84], integrating object-oriented techniques and simplified kernel design to provide high-performance OS support for applications running on a single-level store [4, 107].

MEMORY AND OBJECT MODEL Multics was one of the first systems to use segments to partition memory and support relocation [9, 31]. It used segments to support location independence, but still stored them in a file system, requiring manual linkage rather than the automated linkage in Twizzler. Nonetheless, Multics demonstrated that the use of segmenting for memory management can be a viable approach, though its symbolic addresses were slow.

The core of Twizzler's object space design uses concepts from Opal [19], which used a single virtual address space for all processes on a system, making it easier to share data between programs. However, Opal was a single-address space OS, which is insufficient for a full data-centric system[39], and, while it resulted in a speedup of data transfer and sharing as well as interfacing with devices, it did not address issues of file storage and name resolution. It also still required a file system, since there was no way to have a pointer refer to an object with changing identity, whereas our approach of late-binding for pointers removes the need for an explicit file system. Other single-address space OSes, such as Mungi [54], Nemesis [101], and Sombrero [115], show that single address spaces have merit, but, like Opal, do not fully align with software and hardware trends today; in particular, how the use of fixed addresses results in a great deal of coordination that is unnecessary in our approach. OSes such as HYDRA [124] provide functionality similar to invariant pointers; however, in Twizzler, we extend their use from procedures-referencing-data to a more general approach. Furthermore, they required heavy kernel involvement, an approach incompatible with our design goals.

[39] As we will soon see.

Single-level stores [33, 106, 109] remove the memory versus persistent storage distinction, using a single model for data at all levels. While well-known, "little has appeared about them in the public literature" [106], even since the EROS paper. Our work is partially inspired by Grasshopper [33], AS/400, and orthogonal persistence systems, but while these are designed to provide an illusion of persistent memory, Twizzler is built for both real NVM and distributed sharing, and focuses on providing a truly global object space with global references without cross-machine coordination. Clouds [32] implemented a distributed object store in which objects contained code, persistent data, and both volatile and persistent heaps. Our approach uses lighter-weight objects, allowing direct access to objects from outside, unlike Clouds. Software persistent memory [51], designed to operate within the constraints of existing systems, built a persistent pointer system using explicit serialization without cross-object references, in contrast to Twizzler.

Recently, several projects have considered the impact of non-volatile memories on OS structure. Bailey, *et al.* [4] suggest a single-level store design. Faraboschi, *et al.* [46] discuss challenges and inevitable system organization arising from large NVM, and we follow many of their recommendations. The Moneta project [16] noted that removing the heavyweight OS stack dramatically improved performance. While Moneta focused on I/O performance, not on rethinking the system stack, we leverage their approach to reduce OS overhead as much as possible, even when the OS must intervene. Lee and Won [76] considered the impact of NVM on system initialization by addressing the issue of system boot as a way to restore the system to a known state; we may need to include similar techniques to address the problem of system corruption.

OBJECT MODEL IBM's K42 [73] inspired the high level design of Twizzler. The object-oriented approach to designing a micro or exokernel used in K42 is an efficient design for implementing modular OS components. Like K42, Twizzler lazily maps in only the resources that an application *needs* to execute. Similar techniques for faulting-in objects at run-time have been studied [57]. Communication between objects in Twizzler is, in part, implemented as protected calls, similar to K42.

Emerald [65, 66] and Mesos [55] implemented networked object mobility, which we can also support. Emerald implemented a kernel, language, and compiler to allow objects mobility using wrapper data structures to track metadata and presenting objects in an object-oriented language, impacting performance via added indirection for even simple operations.

The Twizzler object model was shaped by NV-heaps [24], which provides memory-safe persistent objects suitable for NVM and describes safety pitfalls in providing direct access to NVM. While they have language primitives to enable persistent structures, Twizzler provides a lower-level and uninhibited view of objects like Mnemosyne [122], allowing more powerful programs to be built. Languages and libraries may impose further restrictions on NVM use, but Twizzler itself does not. Furthermore, Twizzler's cross-object pointers allow external data references by code, whereas NV-heap's and DSPM's [105] pointers are only internal. Existing work beyond Multics on external references shows and recommends hardware support [102, 123], but provides a static or per-process view of objects, unlike Twizzler, limiting scalability and flexibility.

Projects such as PMFS [38] and NOVA [126] provide a file system for NVM. Twizzler, in contrast, provides direct NVM access atop of a key-value interface of objects. Although Twizzler does not supply a file system, one can be built atop it. While NOVA and PMFS provide direct access to NVM, NOVA adds indirection with copies. Both use `mmap` (which falls short as discussed above) and, unlike Twizzler, require significant kernel interaction when using persistent memory.

Our kernel that "gets out of the way" is influenced by systems such as Exokernel [44] and SPIN [10], both of which drew on Mach [1]. In Exokernel, much of the OS is implemented in userspace, with the kernel providing only resource protection. Our approach is similar in some respects, but goes further in providing a single unified namespace for all objects, making it simpler to develop programs that can leverage shared, in-memory state that can nonetheless persist. In contrast, SPIN used type-safe languages to provide protection and extensibility; our approach cannot rely upon language-provided type safety since we want to provide a general purpose platform.

4.4.1 *Classifying Operating Systems*

Broadly speaking, operating systems can be classified several ways—how they handle and abstract persistence, how they handle inter-process communication, how they access data objects, and how they protect data and processes. We will be looking primarily at single-level stores, since such an interface is a natural fit for in-memory data structures.

4.4.1.1 *Single Level Stores and Single Address Space Operating Systems*

Closer persistence, and NVM in particular, allows the implementation of a *true* single-level store, as has been suggested before [4]. Classic single-level store systems, such as AS/400, Cricket [109], Grasshopper [34], and EROS [106], hide the traditional two-level storage hierarchy of DRAM and disk behind the illusion that all data is in memory. Since these systems present merely the illusion of persistence through memory, they can be broken up into how they provide that illusion. AS/400, while presenting a single-level store interface for data access and manipulation, requires explicit OS calls to ensure persistence of data, while the others provide implicit persistence [34]. Hardware trends indicate that explicit kernel calls to persist data are unacceptable due to their latency[40]. The other systems follow different strategies for implementing implicit persistence, including checkpoints to disk in EROS and completely invisible persistence in Grasshopper. Both of these approaches are inappropriate for a data-centric system, since consistency must be more fine-grained than checkpoints, and require *some* application involvement.

[40] And implicit kernel-provided persistence without application control can result in hazards [29].

Single address space operating systems (SASOSs) and single-level stores are fundamentally entwined, where single-level stores are made easier to implement in a SASOS style, and SASOSs typically present a single-level store interface. Opal [18], Mungi [54], and Sombrero [86] are built for large virtual address spaces, and tie persistent objects to virtual addresses for the duration of their lifetimes. This approach has merit for implementing invariant pointers, but falls short in some respects. Firstly, modern CPU address spaces are not large enough to address all of the data as object storage grows and scales beyond a single machine without increasing pointer size and page-table depth, both of which we would like to avoid. Secondly, the management cost of persisting the virtual mappings outweigh the benefits; since pointers are stored directly, changing the address space (and therefore updating all pointers associated with deleted or moved objects) becomes intractable. Opal, in addition, still required a filesystem, which is a needless layer of abstraction handled within the OS. Other SASOSs [54, 86, 101] also do not take into account NVM, and so do not address the issues described above. HYDRA [124] also provides similar location independent pointer references, but these were used primarily for procedures referencing data, not references to persistent or shared data, and required heavy kernel involvement.

4.4.1.2 *Micro, Exo, Multi, oh my!*

Fundamental research into kernel design plays a large role in our design decisions. The three kernel design philosophies we will discuss are Microkernel [1], Exokernel [44], and Multikernel [6]:

- The *Microkernel* approach limits the responsibilities of the kernel to providing thread scheduling, process separation, and message passing primitives. It allows userspace servers to implement most of the OS functionality applications require. While this is advantageous in moving kernel functionality into userspace, it often does so by message-passing. We have the ability to do IPC on persistent objects through shared memory, so our systems should rely on this approach instead. Additionally, message passing in microkernels typically involves numerous kernel invocations, which hamper persistent and distributed data access.

- *Exokernels* avoid as much operating system abstraction as possible, choosing instead to only handle the bare minimum responsibilities needed for securely multiplexing in-kernel. Exokernels typically involve a user-space *library operating system* (`libos`) that provides other functionality. Unlike the microkernel approach, these often involve procedure-based IPC [44, 74] and present raw *physical* resources to userspace instead of virtualized ones. While the reduced level of message passing is advantageous in our expected architecture, the direct presentation of physical resources to applications presents an unnecessary complexity for persistent data access, and the lack of abstraction for hardware is similarly costly for security and ease of system programming.

- The *Multikernel* approach [6] considers each NUMA domain or each individual core as separate and running a full kernel, with little shared state between instances. While this approach has merit for the distributed nature of a single machine, it does not consider how access to persistent, shared memory could improve application and system design and performance. We take this approach into consideration in our design. However, our above observations suggest that the multikernel approach does not go far enough—*every* programmable device in this system needs to be treated as a separate system instance. Finally, multikernels do not address making hardware see a uniform object space, nor do they address the problem of invariant pointers in a global address space.

4.4.1.3 *Access Control and Capability Systems*

Our discussion thus far has not fully addressed the issue of *data protection*. In a world where hardware devices are free to act on shared global memory with the autonomy that we predict here, protecting data is a significant issue from the perspective of OS design. So far, the hardware solutions available to us for implementing access to shared memory enable protection as well though the IOMMU, whose original design was to both protect memory from hardware and enable full device virtualization [83]. We can leverage this hardware to apply security enforcement as well as a coordinated mapping of objects within a fault zone.

While the enforcement is done through hardware, not the operating system[41], we can still separate mechanism and policy. The users and system make the policy, while the OS interprets the policy, verifies it against a set of requirements and properties, and then programs the hardware to enforce the guarantees. This means the kernel must have a way of verifying access control to an object during a fault. We look to capability systems for solutions due to their common use in single-level stores and SASOSs, and because they provide better least-access properties [87], which is important when hardware devices and applications can cause irreparable damage to persistent data with normal memory accesses.

[41] A necessary consequence of removing the kernel from the data path!

4.5 CONCLUSION

The data-centric approach has the potential to improve performance, simplicity, and efficiency. The design space we described in this chapter is large, but at its core, it has a simple idea: data is the center of programming, and placing data in a global address space can alleviate the context problem. While prior OS research has approached some of these ideas, rarely are they all combined and viewed alongside with our analysis of modern hardware trends. Now that we have gotten the trends, motivation, and backstory out of the way, the next few chapters will focus on design and implementation details for Twizzler.

5 A GLOBAL ADDRESS SPACE

Hollowed out,
clay makes a pot.
Where the pot's not
is where it's useful.
Cut doors and windows
to make a room.
Where the room isn't,
there's room for you.
So the profit in what is
is in the use of what isn't.

—*Tao Te Ching*, Lao Tzu,
translation by Ursula K. Le Guin

SYNOPSIS *This chapter will describe the method by which Twizzler manages a global namespace, starting with how we define memory objects, how we name data within the space, and how we avoid coordination when assigning object IDs. We will then discuss object ID collision, and the probabilistic arguments therein. Finally, we will discuss two implementation details about how we provide software and hardware access to data in the global address space on current hardware.*

Constructing a global address space of all data means that any piece of data must be, in principle[42], nameable from any context. As such, any identification of data will need fairly long names if it is to contain all data within a possibly large system. It is important to note here that I am using the term "name" to refer to a single unique identity within a global, authoritative naming scheme. I am not talking about, for example, C strings or a path. In addition to naming data, a global address space will need a mechanism to ensure that names are in fact unique.

[42] The ability to name a piece of data is disconnected from the *rights* to access that data. Furthermore, *discovery* of a name can also limit an application's ability to name data in practice, even if in theory it can name any data in the address space. These topics will be discussed further in Section 7.3.

5.1 MEMORY OBJECTS

To address these problems, Twizzler organizes data into *objects*. Each object is identified by a unique 128 bit *object ID*. Objects provide contiguous regions of memory that organize semantically related data with similar lifetimes and permissions. Applications access objects via mapping services (discussed later in this chapter) by mapping each object into a contiguous range in the address space, though the address space itself may be densely or sparsely mapped. Objects can be anywhere from 4 KiB, the size of a page, to 1 GiB; the upper bound on object size is an implementation choice, and not fundamental to the design.

An object, from a programmer's perspective, is flexible in its contents—for example, it could contain anywhere from a single B-tree node to the entire B-tree. Often, an object would contain the entire tree, since the entire tree is typically subject to the same access semantics by programs, and there are overheads associated with objects that can be amortized over larger spaces. Data and data structures that are too large for one object or require different access permissions can span multiple objects with references between them.

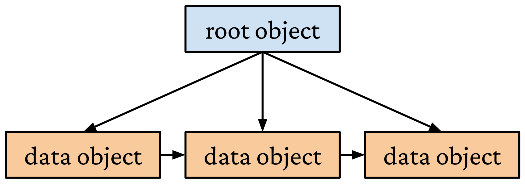

FIGURE 5.1
Using a root object as an index to list multiple data objects that can be viewed logically as a contiguous block of data.

Via objects, we gain a mechanism to name data. Since objects have a maximum size, we can simply index into them at a byte level and form a name of a piece of data within object X as a tuple of (X, n) for an n byte offset into the object. If all object IDs are unique, which we will discuss later in this chapter, then the tuple fits our requirement for an authoritative name.

One might question the decision to name data via a byte offset instead of a richer set of semantics. Our reasoning is based around the level of the system we are currently working with. The basic construction of a global address space with fixed-size object IDs and offsets is designed to be an *underlying component* of a richer, higher level API. Indeed, even our notion of invariant references that we will discuss in the next chapter is a "low level" mechanism. Nothing prevents higher-level APIs and models from being built atop our lower level models—in fact, we plan for that! The low level address space construction is designed to be useable across a variety of languages and environments, not to mention its intended use by hardware devices.

OBJECTS EXPRESS LOCALITY The primary reason that Twizzler organizes the global address space into objects instead of simply providing a large, 192 bit[43] address space for all data is that it is useful to chunk the address space. We will see one major application of chunking in the next chapter (invariant pointers), but there are several other reasons we use objects. First, allowing the kernel to operate on objects instead of byte ranges dramatically simplifies the implementation and allows APIs and services to operate on whole objects at a time. One can think of this as the same underlying reason that virtual memory mappings operate on pages and not on bytes.

[43] This number comes from the object ID size (128 bits) plus 64 bits (for alignment) of an offset into an object.

Objects additionally enable the programmer to implement a direct, logical expression of locality. The ability to group data as either semantically related, access-rights-equivalent, or as "likely to be used at the same time" is fundamental to systems design. Programmers often group data for performance reasons like improving cache hit rate, or choose explicitly to *separate* items to avoid false sharing. Providing a logical expression of locality through objects enables these kinds of designs that programmers often choose.

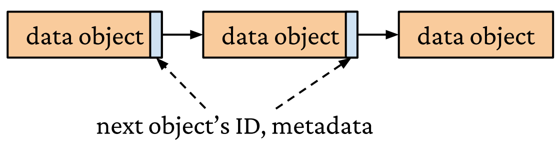

FIGURE 5.2
Similar goals as in Figure 5.1, but here we do away with the index object.

CHAINING OBJECTS The 1 GiB maximum object size may seem quite limited, but this may be because it is natural to think that they are Twizzler's analogue to files in Unix. This is not the case—they are more similar to pages. If one wanted a file abstraction in Twizzler, wherein the size of the file can grow largely without bound, we can simply chain multiple objects together. Figures 5.1 and 5.2 show some possible schemes for accomplishing such chaining, wherein we either form a linked list of objects (which would be effective for small, linear-accessed objects) or have an index object that provides a list of data object for this "file". A variety of mechanisms are possible, ranging from those depicted here, to nesting such solutions and forming an "indirect block" style of tracking data. Since the minimum object's *physical* size is 4 KiB, the overhead of having extra indexing objects is low.

5.2 OBJECT IDS

We already discussed above that object IDs are 128 bits in size, but we should take a moment to discuss this choice, along with how IDs are allocated. One aspect of our global address space we haven't discussed is coordination on IDs—that is, when creating a new object, we must assign it an ID, but where does that ID come from?

One solution is via a centralized mechanism—some server hands out IDs when asked. The trouble here is scalability: that centralized server will be quickly overloaded. Furthermore, the stated aim of our system is to allow devices and nodes to act more independently to enable more concurrency. Yet a centralized solution would force them all through this single bottleneck. Another similar solution would be a federated ID system, where we allocate ranges of IDs to a set of centralized allocators, similar to MAC addresses. While a federated approach alleviates the problem somewhat, it still has scalability problems and requires some out of band coordination ahead of time.

Part of the reason we selected a large ID space in the first place is to make it possible to assign IDs without *any* coordination. To ensure object ID uniqueness, we assign IDs by randomness. As long as we can ensure a high enough chance of avoiding collision in the ID space, this method is scalable and fast.

We employ a probabilistic argument for avoiding collisions in the object ID space. Since object IDs are generated randomly, we can model object IDs as an occupancy problem in a space of $N = 2^{128}$ bins. Thus the probability of collision is [88]:

$$1 - e^{-\overbrace{m\,(m}^{\text{\# of objects}}-1)/2\underbrace{N}_{\text{\# of object IDs}}}$$

If a system were to create millions of objects per second over the course of hundreds of years, the probability of ID space collision is approximately 1 in 10^7. However, such a system is all but guaranteed to suffer hardware errors that flip bits[44] in that time frame, and so an ID space collision is dramatically more likely to be sourced from a hardware malfunction[45]. Further, the ID space could be expanded to 256 bits, which would dramatically reduce the chances of collision.

[44] Take DRAM bit error from cosmic rays, for example.

[45] This is a common comparison when considering probability of a randomized computer process [56].

The reliance on randomized ID allocation is the primary reason we chose to make our object ID space so much larger than some previous approaches [5, 97]. Consider that if $N = 2^{64}$ bins, the probability of collision with just one billion objects is around 2%. Increasing m to one object per person on earth[46] brings the collision probability up to nearly 75%. Of course such an ID space is usable if the ID are allocated via coordination with a centralized allocator, but if we wish to allow object ID creation at scale, avoiding coordination is vital[47]. These alternative approaches with smaller ID spaces lead to problems when sharing, since moving an object from one node to another requires either coordination or application-specific fixup operations to avoid collisions.

[46] *At time of writing*, anyway.

[47] As we will discuss in the next chapter, some of these previous approches chose to put up with coordination and centralization because of the way they encoded their references. Twizzler uses a different method that allows the ID space to be much larger.

5.3 MAPPING BACK TO VIRTUAL MEMORY

While virtual addresses are the wrong abstraction for data access in a data-centric system, modern hardware provides (and often requires) the use of virtual address hardware that we can leverage for protection and isolation, adding additional ephemeral state. Often these virtual address spaces are the only method for applications to access memory. Thus Twizzler uses virtual memory hardware to provide a data object access mechanism to programs.

Twizzler defines objects called "views", which define the current layout of the virtual address space[48]. Twizzler provides access to data objects by mapping them into the virtual address space behind-the-scenes (via `libtwz`). The view object contains structures to define the layout of the virtual address space which the kernel reads and uses to program the MMU accordingly.

[48] Among other things, see Chapter 7.

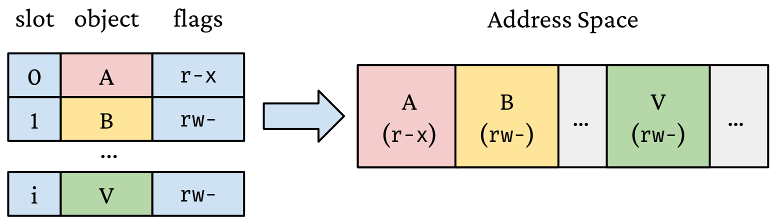

FIGURE 5.3
Layout of a view object. The kernel consults the view object's mapping on page-fault, and maps in the requested objects at the appropriate location in the virtual address space.

Figure 5.3 shows how view objects lay out the address space of any threads running inside that particular view. View objects are manipulated by userspace and interpreted by the kernel. When applications map objects, they update the view to specify that that object should be addressable at a specific location. On a page-fault, the kernel reads the view and maps the object at the requested location. The view object is laid out like a page table, where each entry in the table corresponds to a slot in the virtual address space. Each table entry contains an object ID and requested protection bits to further protect objects atop access control mechanisms, similar to `PROT_*` in `mmap`.

When a page-fault occurs, the fault handler tries to handle the fault by either doing copy-on-write, checking permissions, or by trying to map an object into a slot if the view object requested one. If it cannot handle the fault, *e.g.* due to a protection error or an empty entry in the view object, it elevates the fault to userspace where `libtwz` handles it, possibly by killing the thread, or possibly by mapping an object if the slot is "on-demand". This is similar to userspace paging systems [1, 53]. When the kernel maps an object into a slot, it updates the address space's page tables appropriately.

Note that the use of virtual memory hardware and temporary object mapping into an ephemeral space does not mean that we aren't using a global address space. In fact, in the next section we'll discuss *another* non-global address space that we use. Instead, think of these address spaces as implementation quirks for specific hardware. On x86, the use of virtual addressing is all but required—if we had other hardware and addressing designs, we would have a different layer here for providing actual object access. Finally, applications can ignore most of the `libtwz` mapping work, since they still refer to data via the object ID and offset tuple.

5.4 A LOGICAL ADDRESS SPACE

The interface presented to hardware must enable interaction with a *heterogeneous memory system* and support the abstraction of an object space rather than a collection of flat memory spaces. Unlike applications, hardware has no need for a global address space. Rather, hardware must have the ability to transfer large chunks of objects to, from, and around memory, access memory words, and must be able to handle access to different *types* of memory, preferably in a way that is largely transparent to applications.

It's important to note that here when I'm talking about *hardware*, I am drawing an important distinction between the software (or firmware) running on a hardware device, and the hardware itself. The former I'm grouping in with software in general, as in the limit, we want that software or firmware to interact with data and other applications just as any other piece of software might. By contrast, the hardware is the substrate upon which the software runs. Hence while software sees and interacts with a global address space, the hardware sees but wires coming out to electrically interact with other components.

Both software and hardware must have a way of translating a name for some piece of data into a physical address. The software's method is to rely on mapping functionality provided by the hardware, thus kicking the can down the road. The hardware must therefore know how to translate an object ID and offset into physical memory. This mapping may change frequently as the OS changes allocation of physical pages to data (*e.g.*, to persist a piece of data). Hardware need only know the mappings for a short time, and need not even care about the "canonical" name for the object or piece of data—it simply needs to know how to access data in memory for a single operation. In contrast, software must have longer-term mappings, and must be able to support data shared between threads, potentially mapped into the threads in different places. Our design must support appropriate abstractions for both software and hardware, allowing software to name data while providing hardware and the operating system with the ability to move that data into, out of, and around physical memory, preferably with compatibility for existing hardware functionality.

We use an additional abstraction that sits underneath the global address space—a *logical address space*. The logical address space allows hardware to address data without needing knowledge of the data's physical location. Today, hardware must worry about emitting loads and stores to physical memory, but in a world of heterogeneous physical memory, the location of data is likely to change overtime. Ideally, this movement will be transparent to software, with the exception of setting policy, but for hardware, it cannot be. Requiring hardware to access a global object space through a $\langle object, offset \rangle$ tuple is unnecessary overhead and complexity—hardware need only ever access the "working set" of objects currently undergoing computation. Thus, the logical address space maps objects within it to physical memory pages, managed by the operating system. This solves the heterogeneity problem by allowing hardware to

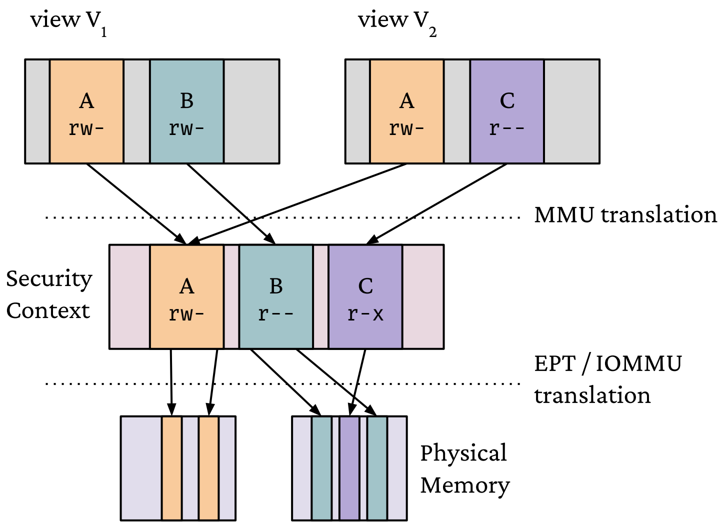

FIGURE 5.4
Two-level translation scheme. The top level maps objects in virtual memory (defined by view objects) to those in object-logical space. The layout of the second level address space is managed by the kernel, since it is not user-facing, and the permissions are derived from a security context object. The second level maps to physical memory.

refer to data within objects instead of physical addresses, whose lifetimes are much smaller than the objects, and doesn't require hardware to emit (likely large) addresses for the global address space.

While SASOSes are not a viable solution to the problem of persistent pointers, they are *a* solution to implementing the logical object space. Hardware, and the CPU, are directly connected, reducing the cost of invalidation and coordination of an address space. Additionally, this address space is *intermediate* and hidden from programs—a virtual address is translated to the logical object space, after which the address is translated to a physical location (shown in Figure 5.4) by a second address translation. The result is that all devices, including the lower level parts of the CPU, can share a single map of current objects in the logical object space. We implement this on current x86 hardware via a combination of the IOMMU and the Extended Page Tables, and will discuss this in more detail and how we use it to enhance security and low level kernel implementation in Chapter 7.

5.5 CONCLUSION

A global address space provides a way for us to organize all data, and giving all data an authoritative name via an object ID and an offset allows us to provide a substrate for building more complex semantics later. Importantly, the uniqueness of object IDs and the isolated process for generating them means that we can avoid coordination on organizing the global ID space. As we will see, the choices of a large ID space that needs little coordination will play a big part of our solution to distributed memory. But now we are left with a question: if we name data via a tuple of an object ID and an offset, how do we implement an actual pointer?

6 INVARIANT REFERENCES

All problems in computer science can be solved by another level of indirection.

—David Wheeler

...except the problem of too much indirection.

SYNOPSIS *This chapter builds up the notion of data references within the global address space discussed in the previous chapter by way of introducing the key abstraction that underpins Twizzler—the invariant reference. We will discuss the design and implementation of invariant references along with several case studies and performance analyses.*

Last chapter, we discussed how we can name any piece of data within the global address space using a tuple of object ID and offset. However, with a large object ID space and an offset value, the tuple is quite big (192 bits). If we want to encode a reference to a piece of data, any pointer needs to encode all that information. But our current virtual address pointers are 64 bits wide, so we would be increasing our overhead over virtual addresses by a factor of 3. Instead, Twizzler uses an *invariant pointer* abstraction that reduces overhead and maintains pointer sizes at 64 bits while making it possible to refer to any data in the global address space (thus, these pointers are inherently cross-object or inter-object), and exposing object relationships to lower level parts of the system stack that would traditionally be blind to such relationships.

6.1 IMPLEMENTATION

The authoritative name for a piece of data as defined in the previous chapter is,

$$\left(\overbrace{I_O}^{\text{ID of object } O}, \underbrace{N}_{\text{An offset into an object}}\right)$$

To efficiently encode this tuple, we use indirection through a per-object *foreign object table* (FOT), located at a known offset within each object. The FOT is an array of entries that each stores an object ID, or a name that resolves into an object ID, as we will see below, and flags. An invariant pointer is thus a tuple,

$$\left(\overbrace{f}^{\text{Index into the FOT}}, \underbrace{N}_{\text{An offset into an object}}\right)$$

and an FOT entry is also a tuple,

$$\Big(\overbrace{I_T}^{\text{Target object ID}}, \underbrace{\text{flags}}_{\text{Flags for this FOT entry}}\Big)$$

The invariant pointer tuple is encoded into 64 bits by placing f into the upper 24 bits and placing N into the lower bits, and the FOT entry tuple is 256 bits wide. We can then recover the original tuple by performing an FOT lookup. For example, if we have an invariant pointer in object O, the lookup is:

$$\Big(\underbrace{I_T}_{\text{Target object ID}}, N\Big) = \Big(\overbrace{F_O}^{\text{FOT Index Operation}}(\overbrace{f}^{\text{Index into the FOT}}), \underbrace{N}_{\text{An offset into an object}}\Big) \tag{6.1}$$

This provides us with both large offsets *and* large object IDs, since the IDs are not stored within the pointer itself. If an object wishes to point to data within itself (an *intra-object* pointer), it stores $f = 0$ in the invariant pointer. When dereferencing, Twizzler uses f as an index into the FOT, retrieving an object ID. The combination of a FOT and an invariant pointer logically forms an authoritative name, as shown in Figure 6.1 and Equation 6.1. Since the FOT is a per-object structure and is stored within that object, objects are *self-contained* and all context needed to interpret pointers in an object is stored within that object.

Our design differs from existing frameworks [5, 9, 20, 27, 31, 102] because of the indirection. Frameworks like PMDK store entire object IDs within pointers, increasing pointer size and reducing flexibility by removing the possibility of late-binding (discussed below). Additionally, Twizzler extends the namespace of data objects beyond one machine, as machine-independent data references are a natural consequence of cross-object pointers. Existing solutions are limited in this scalability. They either limit the ID space (necessary for storing IDs in pointers) and thus resort to complex coordination or serialization when sharing, or they require additional state (*e.g.* per-process or per-machine ID tables) that must be shared along with the data, forcing the receiving machine to "fix-up" references. Worse still, the fix-up is application-specific, since the object IDs are within any pointer, not in a generically known location. Our per-object FOT results in self-contained objects that are easier to share, thus interacting better with remote shared memory systems.

Part of our motivation for indirecting pointers through the FOT was to allow a large ID space without increasing pointer size. The density of NVM and the disaggregation of memory and applications means that we will be accessing data in a larger and larger address space, and it is vital that our abstractions allow for a large enough ID space to

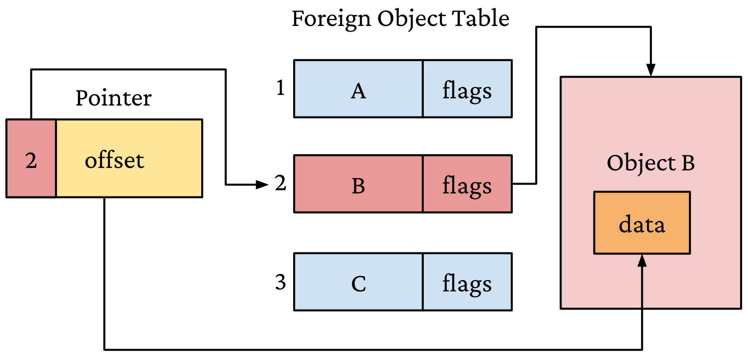

FIGURE 6.1
Pointer translation via the FOT. The pointer and the FOT are both contained in the same object (not shown). An FOT entry of 0 indicates an "internal" pointer.

cover these needs. Since our IDs are 128 bits and our offsets need to support large objects, replacing pointers with a "fat pointer" style of just `object-id:offset` would mean more than doubling pointer size, which we found unacceptable. Other frameworks like PMDK, by contrast, increase pointer size to 128 bits for each pointer by encoding pointers as this tuple with 64 bit object IDs. The trade off is that our pointers take a little more work to translate (as they require an FOT lookup), but in return we keep pointers 64 bits while supporting a truly global-scale address space. We will discuss space overhead later in this chapter.

FLAGS The FOT entry's `flags` field has bits for read, write, and execute protections. The protections are *requests*; Twizzler implements separate access control on objects. This allows some pointers to refer to data with a read-only reference while others can be used for writing, reducing stray writes (a single ID can repeat in the FOT with different protections). The FOT entries also enable atomic updates that apply to all pointers using that FOT entry.

LATE-BINDING Instead of *requiring* programmers to refer to objects via IDs only, we allow names in FOT entries. These entries may contain a pointer to an in-object string table that contains a name[49]. Names enable late-binding [31], a vital aspect of systems, allowing references to objects which change over time, *e.g.* shared library versions. Names are passed to a *resolving* function, specified in the FOT entry. Allowing a program to specify how its names are resolved increases the flexibility of the system beyond supporting UNIX paths.

[49] The name pointer is packed into the FOT entry by replacing the target object ID and setting a bit in the flags field to indicate that this entry contains a name.

The implementation of naming is orthogonal to Twizzler's design. We allow a range of name resolution methods within the system stack and allow objects to specify their own name resolution functions for flexibility. For example, objects could be organized by both a relational database and a hierarchical namer similar to conventional file systems. Non-hierarchical file systems are well studied [2, 48, 49, 95, 96], but these

```rust
struct Hdr {
    inv_ptr: IPtr<i32>,
}

let vptr = hdr.inv_ptr.lea();
hdr.inv_ptr.store(vptr);
// *hdr.inv_ptr is also sufficient for dereference.
```

FIGURE 6.2

Example Rust code that converts an invariant pointer to a dereferenceable form and back.

```c
struct Hdr {
    int *inv_ptr;
}

int *vptr = ptr_lea(o, hdr->inv_ptr);
hdr->inv_ptr = ptr_store(o, vptr);
```

FIGURE 6.3

Example C code that converts an invariant pointer to a dereferenceable form and back.

systems do not easily cooperate atop a single data space. Since Twizzler uses a flat namespace as its "native" object naming scheme, it enables the required cooperation.

POINTER TRANSLATION API Twizzler programs use pointer translation helper functions to handle the translations discussed above. The particular API depends on the language used, and, while some languages require translation functions to be written manually, these can be emitted automatically with proper compiler support. When an invariant pointer must be translated, the program performs the operation shown in Equation 6.1 to convert the pointer into a virtual address, and a reverse operation to convert a virtual address into an invariant pointer. The results can be cached to reduce translation calls. These functions efficiently detect if a pointer needs no translation, allowing the compiler to emit calls to these functions for pointers whose origin it does not know.

For an example in the C API, shown in Figure 6.3, if we have an object (o), and a structure in the object containing a pointer to data, we use `ptr_lea` to convert it into a current virtual address. This call returns a virtual address that points to the requested data whether `hdr->inv_ptr` is an intra-object or cross-object pointer. A similar operation on UNIX would require calls to `open`, manually written code to store a path to the target object (whether or not it is different from o), and additional calls to `read` or `mmap` to get the data—effectively reimplementing Twizzler's pointer framework with more complexity.

Writing a cross-object pointer is simple as well, as shown in Figure 6.3. If `vptr` will form a cross-object pointer, an FOT entry will be added to o if needed, and the returned value of `ptr_store` will be an invariant pointer to the object pointed to by `vptr` for storage in o. Doing a similar operation on UNIX would require manual pointer serialization. Optimizations like those discussed above can turn these into no-ops in many cases. The resulting simplicity of programs and enhanced object sharing is discussed later, along with an analysis of performance.

A similar API is shown in Figure 6.2 for Rust, but here we gain the advantages of Rust's type system. Not only is the target type of the invariant pointer known, but it is differentiated from the virtual address that we can convert it to (which has type `&i32`). We will discuss Rust in more detail in Chapter 8, in particular how it can be used to avoid common foot-guns that are present in the C version.

Although several existing persistent memory programming frameworks [20, 102, 123] support similar APIs, they do not support per-object indirection, they lack system-wide support, some increase pointer size and do not always provide type-safe translation functions, and they do not enable transparent use of pointer types in the language. Implementing translations manually involves writing code like

```
    (void *)((uintptr_t)foo + vbase),
```

resulting in brittle code that cannot be easily expanded to include cross-object references.

6.2 EVALUATION

In evaluating Twizzler's invariant pointer design, we investigated both the usability of the invariant pointer model and the more measurable elements such as performance impact and storage overhead. We approached the evaluation in two ways—porting existing software (SQLite [116]) and writing new software for Twizzler. The latter demonstrates the true power of Twizzler's invariant pointer model and allows us to explore better the consequences of our design choices without being constrained by legacy designs. Porting existing software, however, has the advantage of demonstrating the flexibility and generality of the model and environment.

We built two pieces of new software for Twizzler: a hash-table based key-value store (KVS) and a red-black tree data structure. Each has different characteristics and goals, and together they demonstrate the flexibility that Twizzler offers in allowing simple implementation, nearly-free access control, and the ability to directly express complex relationships between objects. We then ported SQLite to Twizzler using our KVS and red-black tree code and evaluated it using a YCSB [26, 45] driver, thus exploring Twizzler's model in a larger, existing program. The new software, along with our SQLite port and microbenchmarks, are evaluated for performance in Section 6.2.6, but let's first consider their development and our experiences with writing software in our model.

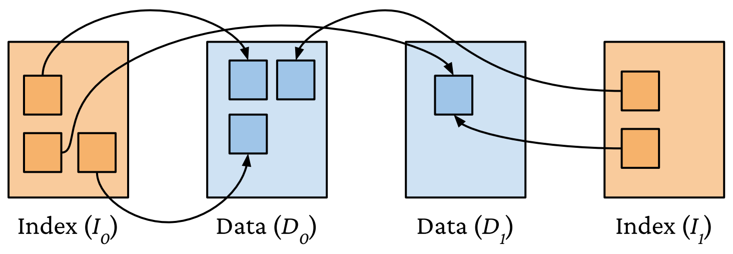

FIGURE 6.4
Cross-object pointers in `twzkv`. The index object contains pointers to keys and values, which can reside in any data object. We also support multiple indexes to enable additional indexing strategies and access control on discovery.

6.2.1 *Case Study: Key-Value Store*

We implemented a multi-threaded hash-table based Key-Value Store (KVS), called `twzkv`, to study cross-object pointers and our late-binding of access control. Our KVS supports insert, lookup, and delete of values by key, both of arbitrary size, and hands out direct pointers to persistent data during lookup. During insert, it copies data into a data region before indexing the inserted key and value. We built `twzkv` in roughly 250 lines of C, in multiple phases to study how our system handles changing requirements. Handing out direct pointers into data was trivial to implement with cross-object pointers, requiring only a call to `ptr_lea` during lookup. The initial implementation maintains two objects, one for data and one for the index. The complexity typically involved when storing both index and data in a single, flat file is not justified in a programming model where we can express inter-object relationships directly at near-zero cost in complexity or performance. In our case, a pointer from the index object to the data object (such as an entry in the hash table) can be written with a single call to `ptr_store`. This, combined with the simple requirements for an in-memory NVM KVS, resulted in a small implementation that was nonetheless a usable KVS.

6.2.1.1 *Extending Requirements*

Next, we added functionality to protect values with access control. We wanted to keep handing out direct pointers to data during lookup and to keep `twzkv` a library (as opposed to a service). Meeting these goals on an existing system would be difficult without adding significant complexity, such as reimplementing a lot of Twizzler's pointer framework or implementing manual, redundant access control.

In Twizzler, implementing access control in `twzkv` involved having the index refer to data in multiple data objects, assigning those objects different access rights, and allocating from those objects depending on desired access rights. We were able to implement this while preserving the original code due to the transparent nature of Twizzler's cross-object pointers. Now, when inserting, the application indicates the data object into which to copy the data, as shown in Figure 6.4.

With multiple data objects, twzkv can leverage the OS's access control, minimizing complexity. Unrestricted data can go in D_0 (Figure 6.4), whereas restricted data can go in D_1. Since each object has distinct access control, a user can set the objects' access rights, then decide where to insert data according to policy. The indexes point to the correct locations regardless of the access restrictions of the data objects, and twzkv still hands out direct pointers, but a user that is restricted from accessing data in D_1 will not be able to dereference the pointer. A further extension is to support secondary indices, as shown in Figure 6.4, enabling alternative lookup methods and limiting data discovery with index object access control. This extension is easy to implement on Twizzler, increasing our implementation's complexity only a small amount.

6.2.1.2 *Comparison to Unix Implementation*

To compare with existing techniques, we built a similar KVS using only Unix features (called unixkv). It also separates index and data, but it must manually compute and construct pointers, requiring a significant amount of programmer time to get right. Supporting multiple data objects was complex in unixkv, because we had to store and process file paths in the index and store references to paths for pointers, increasing overhead and code complexity by 36%—a lot for an implementation with relatively few pointers—just to reimplement Twizzler's support. The extra complexity also included code to manually open, map, and grow files, much of which Twizzler handles internally. Development time was extended by bugs that were not present when developing twzkv arising from the manual pointer processing. While twzkv gains transparent access control, unixkv does not due to the lack of on-demand object mapping and late-binding of security. Instead, unixkv needs to know object permissions before mapping, a restriction that limits the ability to reuse the operating system's access control, something that twzkv could leverage through late-binding on security (Section 7.3)[50]. Other frameworks like PMDK that do not integrate access control and late-binding into their models have similar limitations.

[50] unixkv could trap segmentation faults to do this, but that would be application-specific, difficult, and would reimplement Twizzler functionality.

6.2.2 *Case Study: Red-Black Tree*

To evaluate the process of writing persistent, "pointer-heavy" data structures, we implemented a red-black tree in C using normal pointers (ramrbt) in 100 lines of code, and evolved it for persistent memory in two ways: manually writing base+offset style pointers, as current systems require (unixrbt), and using Twizzler (twzrbt). Porting existing data structure code to persistent memory will be common during the adoption of NVM, and much of the complexity therein comes from dealing with persisting virtual addresses [82].

In developing `unixrbt`, we found 83 locations where we had to perform pointer arithmetic for converting between object addresses and virtual addresses. Consider an expression such as

```
root->left->right = foo.
```

Inserting calls to translate this directly results in

```
L(L(root)->left)->right = C(foo),
```

where `L` converts to a virtual address and `C` converts back, which is heavily obfuscated and took more development time than writing `ramrbt` in the first place due to debugging.

We built `twzrbt` like `unixrbt`, annotating pointer stores and dereferences. However, `unixrbt` used an application-specific solution for pointer management; if other applications wanted to use the data structures created by `unixrbt`, they would have to know the implementation details of the pointer system (or share the implementation, thus reimplementing much of Twizzler's library). Additionally, due to Twizzler enabling improved system-wide support for cross-object pointers, these transformations can be made automatic through compiler support.

Unlike `twzrbt`, `unixrbt`'s tree is limited to a single persistent object; a limitation that prevents the tree from growing arbitrarily, does not allow it to directly encode references to data outside the tree object, and does not gain it the benefits of cross-object data references that were discussed above for `twzkv`. Adding support for this to `unixrbt` would require modifying the core data structures to include paths and significantly altering the code, increasing its length by at least a factor of 2, whereas `twzrbt` gets this functionality for free.

Another advantage of `twzrbt` is reduced support code compared to `unixrbt`; `unixrbt` needed code to manage and grow files and mappings, while we implemented `twzrbt` as simple data structure code with Twizzler managing that complexity. The additional error handling code and pointer validity checks in `unixrbt`—handled automatically in Twizzler—increased development time and implementation complexity.

6.2.3 *Porting SQLite*

We ported SQLite to Twizzler to demonstrate our support for existing software and to evaluate the performance of a SQLite backend designed for Twizzler. We used our POSIX support framework, a combination of `musl` and our library `twix`, to support much of SQLite's POSIX use. We took a modified version of SQLite called SQLightning that replaced SQLite's storage backend with a memory-mapped KVS called LMDB [23]. We chose this port because LMDB is implemented with `mmap`'d files as the primary access method and hands out direct pointers to data as one would expect from an effectively designed NVM KVS[51]. Since LMDB's SQLightning port replaces the storage backend with calls to LMDB, we ported SQLite to Twizzler by taking our KVS and red-black tree code and implementing enough of the LMDB interface for SQLite to run using Twizzler as a backend. Outside of the B-tree source file few changes were needed for SQLite to run on Twizzler. We further ported our modified SQLite backend to PMDK to compare directly with a commonly used NVM programming library that supports persistent pointers.

[51] These are not invariant pointers, however, unlike Twizzler's.

We also ported a C++ YCSB driver [45], which required porting the C++ standard template library (STL). Since we had already ported a standard C library, the C++ STL was easily ported, demonstrating the ease of porting software to Twizzler. We have also ported some existing Unix utilities (such as `bash` and `busybox`), which largely require only recompiling to run on Twizzler. Of course, to gain *all* of the benefits of Twizzler, programs will be need to be written with NVM or data disaggregation in mind (but this is true regardless of the target OS).

Our implementation of the LMDB interface corroborated our experience from the KVS case study: much of the complexity in storage interfaces and implementations comes from the separation between storage and memory. This has been studied before (as we discussed in Section 4.4), but the hardware trends discussed in Chapter 2 change the game significantly by allowing programmers to think directly via in-memory data structures. The result is that interfaces like cursors in a KVS become redundant. We implemented to this interface for LMDB, but the functions were largely wrappers around storing a pointer to a B-tree node and traversing the tree directly without separate loads and copies. The result was an extremely simple implementation (500 lines of code) that still met the required interface. Future software can use Twizzler's programming model to more effectively write software that eschews the need for complexity forced by the two-tier storage hierarchy.

6.2.4 *Case Studies Discussion*

Although these implementations were simple, they represent the applications and data structures we expect in a data-centric system. Invariant pointers we can directly use in our programming languages make computing over persistent data almost transparent, allowing simple implementations that are nevertheless easy to evolve as requirements change.

Not only does `twzkv` have strong support for access control, it enables concurrent use of databases via cross-object pointers. Applications can load indexes for multiple databases without needing to worry about address space layout and without writing complex pointer management code that would be required by an implementation using `mmap`. We were able to provide access control without a single line of code in `twzkv` dedicated to checking or enforcing access rights. Instead, we relied on Twizzler's built-in access control, a reuse not possible with other frameworks that do not support late-binding of access rights and do not consider security as part of their model. Twizzler thus removes the need for applications to enforce and implement their own access control, which increases the security of the system by divesting programmers from the responsibility of getting the enforcement right. Similar functionality for current systems would traditionally require separation of the library and application into a client-server model, but that additional overhead is unneeded here and inappropriate on a persistent memory system.

Although `twzrbt` and `twzkv` had different densities of pointer operations, `twzrbt` being "pointer-heavy" and `twzkv` being "pointer-light", Twizzler improved the complexity of both over manual implementation and improved flexibility over existing persistent pointer methods. Using a system-wide standardized approach to pointer translations not only enables better compiler and hardware support, but it also improves interoperability; because they share a common framework, `twzkv` could use the red-black tree code and data with ease, and even interact with the SQLite database even though they were written separately without that goal in mind. The position-independence afforded by this model enables both composability and concurrency, while also simplifying programming on persistent data to a natural expression of data structures.

6.2.5 *Storage Overhead*

One important consequence of replacing virtual address pointers with invariant pointers is storage overhead. While our design choice to indirect the pointer resolution through a per-object table means we can avoid increasing pointer size[52], we must consider the size of that table when considering the impact of invariant pointers. Of course, while a direct comparison to virtual address pointers is somewhat inappropriate, as virtual addresses are not invariant and thus do not have the same semantics and power, we can use them as a baseline for comparing the increased overhead *factor* of Twizzler versus a "fat pointer" style persistent pointer.

[52] As we will see, this choice is meaningful for performance overhead.

Fat pointers are present in, for example, PMDK [5], which includes an address space of objects with 64 bit IDs, indexed with a 64 bit offset. Thus a pointer in PMDK is 128 bits total, and the overhead relative to virtual addresses is exactly $2\times$. However, PMDK's ID space is much smaller than Twizzler's (64 bits versus 128 bits). As we discussed in Chapter 5, the large ID space is vital and provides useful semantics that a smaller space misses. Adjusting PMDK's pointer model by increasing the ID size brings it to 192 bits per pointer, with a relative overhead[53] of $F_{f128} = 3$.

[53] Here F_{f128} means "the overhead factor of a fat-pointer design with an ID space of 128 bits".

In Twizzler, the overhead relative to virtual addresses is entirely dependent not on the number of pointers, but the number of FOT entries. Each FOT entry is 256 bits in size, so the relative overhead is:

$$\underbrace{F_T}_{\text{Twizzler's overhead factor}} = \frac{64N_p + 256N_f}{64N_p} = 1 + \frac{4\,\overbrace{N_f}^{\text{Total \# of FOT entries}}}{\underbrace{N_p}_{\text{Total \# of pointers}}}$$

While the above relationship is useful for determining what the overhead factor is, the ratio between the number of pointers and the number of FOT entries is application dependent. Some applications will have a large number of references within a small number of objects or will have a lot of intra-object references, and thus will have a small overhead factor, while another application may have a large number of outgoing references per object. Anecdotally, the former has been far more common, which makes sense—objects are a projection of logical locality into an ID space, and thus an application with a large overhead factor is, inherently, structuring its data without significant locality.

However, we can model some real data structures to better understand how their overhead factors are influenced by different elements of data structure organization. Figure 6.5 shows the overhead factors of Twizzler and PMDK (or, more broadly, fat-pointer designs) with different choices for object ID sizes and with different characteristics for data organization. We generated random graphs using a stochastic process that selected edges randomly with a bias towards edges that were between nodes that were close in an ID space. Once the graphs were generated, the nodes were assigned to object IDs in batches. The bias was present to model organizing a graph with locality in-mind.

The generated graphs could be either dense (many edges per node) or sparse (few edges per node), and the objects could be either numerous and hold few objects, or there could be few objects that held many nodes. Figure 6.5 shows that Twizzler's per-FOT overhead is better than the fat-pointer style in most cases, with the exception of having few nodes per object and a spare graph. This correlates with the above equation. Some rearranging shows that to beat the overhead factor of $F_{f128} = 3$ one needs two pointers per FOT entry. When an object has many outgoing references to few objects, this threshold is quickly surpassed.

6.2.6 *Performance*

Our evaluation's primary focus is on the benefits of the programming model, showing new functionality with reduced complexity at an acceptable overhead. Nevertheless, there are many cases where we see significant improvement (such as SQLite) because the programming

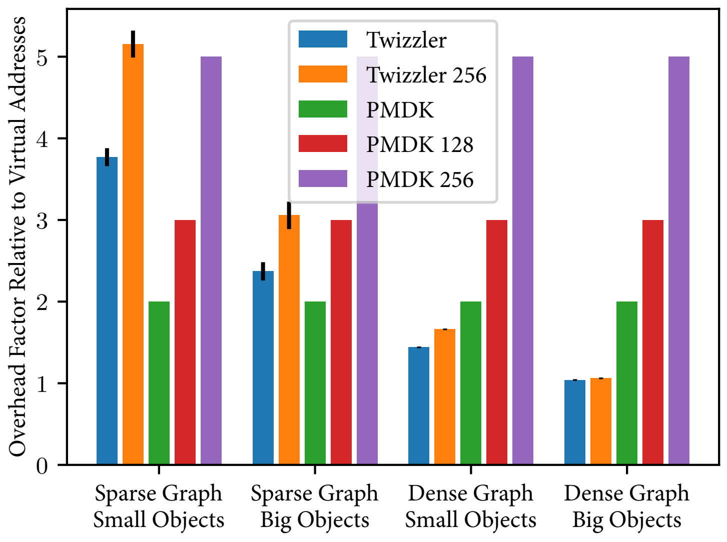

FIGURE 6.5

Storage overhead of Twizzler compared with PMDK at different object ID sizes. PMDK, or fat-pointer style references have a constant overhead per pointer. Note that Twizzler's object ID size is 128 bits, thus the closest comparison is not baseline PMDK, but instead PMDK 128. In most cases, Twizzler has a better overall storage overhead, except in the pessimistic case where many FOT entries are needed.

model has less overhead, and our pointer design is space efficient and fast to translate.

We measured the performance of our KVS and red-black tree, performed microbenchmarks, and evaluated the Twizzler port of SQLite against Linux (Ubuntu 19.10) instances of SQLite, SQLightning, and our port of SQLite to PMDK. Tests ran on an Intel Xeon Gold 5218 CPU running at 2.30 GHz with 192 GB of DRAM and 128 GB of Intel Optane Persistent DIMMs. We compiled all tests against the `musl` C library instead of `glibc` because Twizzler uses `musl` to support UNIX programs.

All Linux tests used the NOVA filesystem [126] (a filesystem optimized for NVM) on the NVDIMMs, mounted in DAX mode. This enabled direct access to the persistent memory without page-cache interposition.

6.2.6.1 *Microbenchmarks*

Table 6.1 shows common Twizzler functions' latencies, including pointer translation (loading and storing) and mapping overhead. The overhead shown for resolving pointers does not include dereferencing the final result, since that is required regardless of how a pointer is resolved. The first row shows the latency for resolving pointers to objects the first time. Twizzler makes a further optimization by caching the results of translations for a given FOT entry. Each successive time that FOT entry is used to resolve a pointer, the result of the original translation is returned immediately, improving the latency as shown on the "cached" row of Table 6.1. Note that the low latency of these results is expected; the performance critical case of these functions' use is repeated calls, and since these operations are simple, they fit within the processor cache.

Pointer Resolution Action	Average Latency (ns)
Uncached FOT translation	27.9 ± 0.1
Cached FOT translation	3.2 ± 0.1
Intra-object translation	0.4 ± 0.1
Inter-object pointer store	17.2 ± 0.6
Intra-object pointer store	2.3 ± 0.1
Mapping object overhead	49.4 ± 0.2

TABLE 6.1

Latency of common Twizzler operations, including pointer loading and storing, and object mapping.

Twizzler translates intra-object pointers by first checking if the pointer is internal and, if so, adding the object's base address to it—the same operation required for application-specific invariant pointers. The expanded programming model offered by Twizzler makes this overhead minor relative to the high costs for persistent data access on current systems, which have high-latency for equivalent operations.

The pointer store operations shown in Table 6.1 measure the latency of the `ptr_store` operation that is used to construct persistent data references in Twizzler. While these operations are less common than pointer loads, their overhead directly affects applications that perform many updates to data structures. The most common pointer store operation applications perform is internal (intra-object) pointer stores, in which the overhead is minimal. Pointer store operations for external (inter-object) references have slightly more overhead, since they need to operate on the FOT.

We compared our pointer translation to Unix functions. Resolving an external pointer with an ID corresponds roughly to a call to `open("id")`, which has a latency of 1036 ± 15 ns. The comparison is not exact, of course; the pointer resolution also maps objects, and the call to `open` must handle file system semantics. However, the direct-access nature of NVM results in pointer translation achieving the same goal as opening a file does today. The pointer operations in Twizzler accomplish much of the same functionality as the heavier-weight I/O system calls on Unix with more utility and less overhead.

A more direct comparison is object mapping, which has low latency compared to `mmap` (658.7 ± 12.7 ns—a $13.3\times$ speedup) though the two have similar functionality. Since mapping occurs entirely in userspace, cache pollution is reduced. While both `mmap` and Twizzler's mapping require page-faults to occur before the data is actually mapped, this overhead is similar in Twizzler and Unix, and so is not shown.

It is important to keep in context the latency of these operations, many of which replace relatively heavy Unix operations in functionality in Twizzler's memory-access model. Most of the equivalent Unix operations require at least one system call, which is untenable for low-latency per-

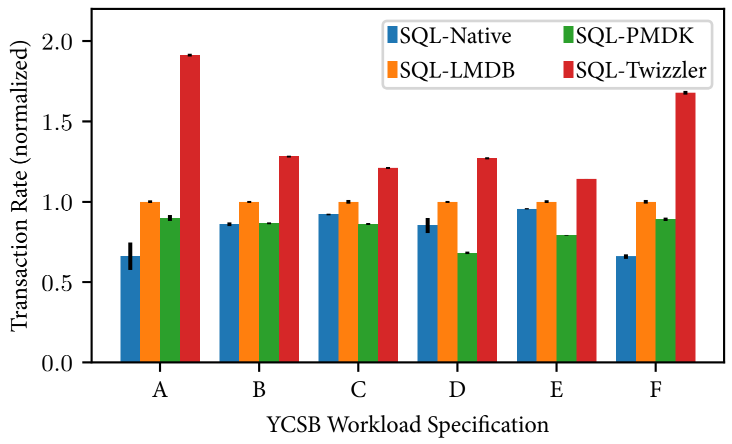

FIGURE 6.6
YCSB throughput, normalized (higher is better). Twizzler out-performs all other variants.

sistent storage. The mitigations for both Spectre [70] and Meltdown [78] further motivate reducing system calls, especially with NVM, since they further increase system call latency.

6.2.6.2 *SQLite*

We ran four variants of SQLite, three on Linux and one on Twizzler, and compared their performance: "SQL-Native" (unmodified SQLite), "SQL-LMDB" (SQLite using LMDB as the storage backend), "SQL-PMDK" (SQLite using our red-black tree on PMDK), and "SQL-Twizzler" (our port of SQLite running on Twizzler). SQL-Native was run in `mmap` mode so that both it and SQL-LMDB used `mmap` to access data. We ran each on the same hardware and normalized the results.

Figure 6.6 shows the three variants' (and baseline) throughput under standard YCSB workloads. The performance improvement of the LMDB and Twizzler variants over SQL-Native is likely due to handing SQLite direct pointers to data. However, in the Twizzler case we get an additional benefit of operating on data structures directly while LMDB has an abstraction cost.

Figure 6.7 shows the latency of queries on a one million row table. This is common data processing—loading and then examining data in a variety of ways. We measured the performance of calculating the mean and median, sorting rows, finding a specific row, building an index, and probing the index. SQL-Twizzler had similar performance to SQL-LMDB and SQL-Native despite comparing its extremely simple storage backend to optimized B-tree backends (that benefit from scan operations). As a more direct comparison, SQL-Twizzler significantly out-performed SQL-PMDK in most tests. PMDK's pointer operations are more expensive than Twizzler's, requiring up to two hash table lookups per translation [5]. Additionally, PMDK's pointers are 128 bits, while Twizzler does not in-

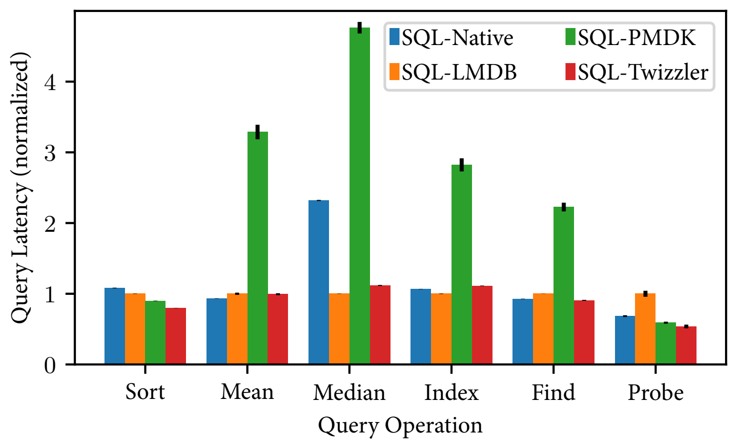

FIGURE 6.7

Query latency, normalized (lower is better). Twizzler maintains a similar level of performance with the native and LMDB variants, despite comparing Twizzler's simplistic red-black tree index implementation with highly optimized B-trees. Twizzler also significantly outperforms PMDK despite sharing a similar implementation for the index structures.

crease pointer size. Increased pointer size results in significantly worse cache performance, especially in a pointer-heavy data structure like a persistent red-black tree.

6.2.6.3 *Key Value Store*

We compared `twzkv` to `unixkv` by inserting one million distinct key-value pairs, followed by looking up each in-order. The inserted items were 32-bit keys and 32-bit values, chosen to reduce the overhead of data copying since we were focusing on pointer translation overhead. Both were compared under two modes, single-data-object and multiple-data-objects. Both KVSes translated between virtual and persistent addresses when storing and retrieving data, but for multiple-data-objects, we allow for storing the data in an arbitrary object.

Figure 6.8 shows the latency of lookup and insert, demonstrating that not only is the memory-based index and data object structure that can hand out direct data pointers sufficiently low latency to take advantage of NVM, but the additional overhead of cross-object pointers is minimal. Compared to `unixkv`, `twzkv` has minimal overhead in the single-object case, and improves lookup performance in the multiple-object case. The minor overhead in other cases comes with improved flexibility, simplicity, and access control support (`unixkv` does not support access control). Finally, multithreaded access on `twzkv` and `unixkv` did not improve performance; despite the pointer translations, they ran at memory bandwidth (for NVM).

6.2.6.4 *Red-Black Tree*

We measured the latency of insert and lookup of 1 million 32-bit integers on both `unixrbt` and `twzrbt`. The insert and lookup latency of `twzrbt` was 528 ± 3 ns and 251.8 ± 0.5 ns, while insert and lookup

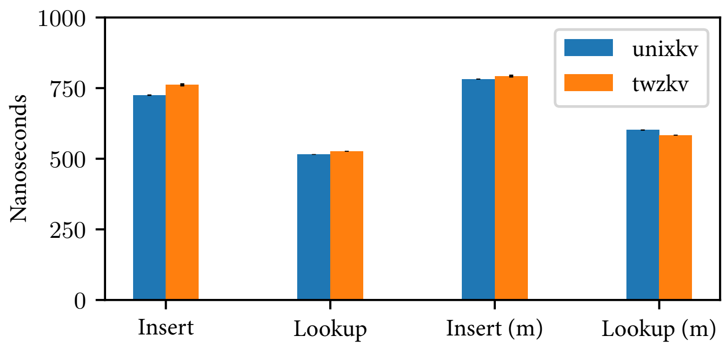

FIGURE 6.8

Latency of insert and lookup in `twzkv` and `unixkv`. An “(m)” indicates support for multiple data objects. Both `unixkv` and `twzkv` have similar latencies.

latency of `unixrbt` was 515 ± 2 ns and 213 ± 1 ns. The modest overhead comes with significantly improved flexibility, as `unixrbt` does not support cross-object trees, and less support code (`unixrbt` manually implements mapping and pointer translations). Note that even though there is lookup overhead in `twzrbt`, this overhead did not predict the results of a larger program—the SQL-Twizzler port used this red-black tree, and saw performance benefits over block-based implementations.

6.3 CONCLUSION

We have discussed the design and implementation of invariant references in Twizzler and demonstrated their power, flexibility, and low overhead. Importantly, the implementation of invariant pointers in Twizzler means that **objects are self-contained** and all context necessary to interpret data in object is stored in that object. However, the ability to create references in a global address space does not a full operating system make. In the coming chapters, we will discuss the OS infrastructure needed to realize our model, along with Twizzler OS services that enable easier programming on the kinds of systems we are envisioning.

7 TWIZZLER: AN IMPLEMENTATION

Operating systems are like underwear—nobody really wants to look at them.

—Bill Joy

SYNOPSIS *While the last two chapters have focused on perhaps the most important parts of Twizzler—global addressing, memory objects, and invariant pointers—there is still importance in understanding some of the other operating system services. This chapter will discuss things like ephemeral state management, controlling objects, threads, and security.*

An operating system provides functions to userspace all in service of enabling the application to operate on data by multiplexing the various pieces of hardware in the system. Twizzler is no different, however it leans towards a lighter hand—more like a microkernel—than, *e.g.*, Unix.

Twizzler is started via a bootloader that loads the kernel and an initial ramdisk (initrd) into memory. The initrd is a simple tar file containing an initial set of memory objects to load. One of these is the init program, which is started by the kernel, and is the only executable that the kernel itself loads. After initializing the system, the kernel starts the init program, which initializes the rest of userspace, including logging, paging, and devices, before starting a shell.

7.1 OBJECT SERVICES

The kernel provides services for object management, including creating and deleting objects. It maintains mapping information, such as which objects are mapped into which address spaces, and common paging tasks like map counts and usage statistics. If directly attached NVM is present, the kernel manages mappings to NVM via a built-in mapping of object pages to NVM pages. While not a filesystem, the kernel's use of NVM does manage allocation and mapping object pages to physical pages.

7.1.1 *Copy-From*

While it is possible for an application to just copy large amounts of data from one object to another via `memcopy`[54], this is inefficient for large data. To provide an efficient large-scale copy operation that takes advantage of copy-on-write functionality, Twizzler exposes a copy-from system call. This system call takes, as arguments, a target object ID and a list of source specifications which contain:

[54] Or a better language's equivalent routines.

1. `srcid`: A source object ID.
2. `src_start`: A starting byte offset, interpreted as an offset into the source object.
3. `dst_start`: A starting byte offset, interpreted as an offset into the target object.
4. `length`: A length for the copy operation.

The kernel then copies data from the source to the target. If it can, it makes use of full-page copies that require only changing mappings (and, for persistent objects, stored object maps), and leverages copy-on-write. If a full-page cannot by copied, the kernel does the byte-level copy on behalf of the thread. Before doing the copy, the kernel locks the target object and the source objects to avoid unexpected changes (though, this behavior can be configured via flags).

7.1.2 *Creation*

Objects are created by the `create` system call, which returns an object ID. The caller can specify two policy-level pieces of information about the object: its *lifetime* and its *backing-type*. The lifetime may be either *persistent* or *volatile*, and the backing type may be any type of physical memory available on the system (or a default). The kernel must adhere to the semantics of persistence or volatility, but may choose any backing type it likes that implements the lifetime requirements[55]—the supplied backing type is merely a hint. In addition to the policy information, the caller may supply a copy-from (see above) list that will fill the new object with data before returning. If the list is empty, the created object contains all zeroed memory, and any areas not covered by the copy-from list will be zeroed. We will discuss in Chapter 8 more details on object lifetime and the safety hazards involved in object creation.

[55] Even storing persistent objects in DRAM and flushing them to stable storage—see below.

7.1.3 *Deletion*

Objects are deleted via the `delete` system call. Like UNIX's `unlink`, objects are reference counted, which includes mappings in an address space. Once the reference count reaches zero, the object may be deleted. During deletion, an object may be optionally marked as "hidden", causing new mapping requests for this object to fail. We will discuss how applications can directly interact with these reference counts in Chapter 8.

7.1.4 *External Paging*

If the kernel is unaware of a requested object or does not have a requested page in core, it will contact the userspace paging service. The pager is started by init early on in the boot process and is allocated a special object for which it shares read/write access with the kernel. This object is called the pager queue, and is used to form a multi-producer, single-consumer submission/completion queue pair. The kernel then communicates with the pager by enqueuing requests and awaiting responses[56]. The pager can then handle paging requests and coordinate with the kernel on eviction. Drivers are handled in userspace in Twizzler, so storage drivers can be implemented as part of the pager via shared libraries for modularity.

[56] The pager also gets a "pager to kernel" communication pathway via a second queue pair.

The userspace pager is also how we can enable persistence and sharing without needing NVM. Not only can the kernel issue requests to the pager to flush pages to stable storage like a traditional microkernel might be architected, but we can enable applications to communicate with the pager to provide requests and hints as to ordering for flushes. Applications can then communicate some transactional semantics to the pager to allow for optimizations. Other strategies are also available, such as using the higher bandwidth of modern SSDs to persist full application state checkpoints at high granularity [121], or persisting objects in a similar way. In all cases, however, the pager can provide mechanisms for the kernel to evict, and for applications to persist, data to stable storage, while maintaining a flexible strategy for doing so that can make use of application semantics to optimize.

7.2 DEALING WITH EPHEMERA

Despite Twizzler's focus on persistent data, many components of our hardware and applications are built around ephemeral constructs. For example, threads are ephemeral "moments of computation" that act on persistent data, while the programs that they execute often expect some ephemeral private data (*e.g.* the data segment and the stack). While virtual addresses are the wrong abstraction for persistent data access, modern hardware provides (and often requires) the use of virtual address hardware that we can leverage for protection and isolation, adding additional ephemeral state.

7.2.1 *Views*

Twizzler defines objects called "views", which coalesce the state and context necessary to support ephemeral constructs like threads and application instances into Twizzler objects. A significant part of that state is

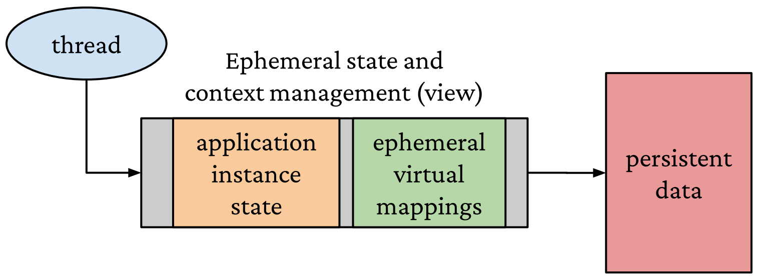
thread
Ephemeral state and context management (view)
application instance state
ephemeral virtual mappings
persistent data

FIGURE 7.1

View objects in Twizzler. Views manage ephemeral constructs and state, giving threads the necessary context to execute and access persistent data.

ephemeral virtual address mappings (discussed in Section 5.3); Twizzler provides access to persistent objects by mapping them into the virtual address space behind-the-scenes (via `libtwz`). The view object contains structures to define the layout of the virtual address space which the kernel reads and uses to program the MMU accordingly. Figure 7.1 shows how views "mesh" ephemeral threads with persistent data by providing them a context to operate in. Since view objects are normal Twizzler objects, they can be persisted, allowing us to recover application state after power cycles.

By coalescing this ephemeral state into an object, we make it possible for applications to manage it directly with minimal kernel involvement. Avoiding the kernel is natural—all data access already does this in Twizzler, so adding a separate kernel API to manage this state would add complexity—and reduces the number of system calls needed when mapping objects. Additionally, avoiding the kernel necessitates an increased address space management responsibility for userspace. For example, executable loading and mapping is largely handled without the kernel.

Applications can add objects to a view with the `view_set` function. The caller specifies a target object and a set of protections (see Chapter 6), and a slot in which to map the object. However, applications rarely invoke this function directly—instead, `libtwz` provides a higher-level API to allow applications to operate above the level of manually mapping objects. The standard library also provides access to other utility functions for views (such as querying state, creating new views, and copying views). These functions, by default, operate on a thread's current view, but they optionally operate on any other view object[57], which allows Twizzler to implement operations with semantics similar to `fork` and `execve`[58].

[57] Given appropriate permissions, of course.

[58] And introspection and debugging!

When threads add entries to a view object they need not inform the kernel—when a fault occurs, the kernel will read the entry as needed. However, when *changing* or *deleting* an entry, threads must inform the kernel so it can update existing page table entries. We provide two system calls for views. The `become` system call allows a thread to change to a new view, which might be used to execute a new program or jump across programs to, for example, accomplish a protected task. Twizzler's access control system prevents this from happening arbitrarily. The second system call is `invalidate_view`, which lets a thread inform the kernel of changed or deleted entries.

View objects not only reduce kernel boundary crossings, but they also improve the resumability of the system. After a power cycle, the OS now has information on which objects were mapped and where, improving the ability of threads to pick up where they left off. Additionally, view objects facilitate the sharing of address spaces between threads, since they can both synchronize on modifying a given view object and need not duplicate information. Note that the particular contents of a view object are system-specific. On virtual memory systems, one of their jobs is to manage ephemeral virtual mappings, while on other architectures one of their jobs may be to manage, *e.g.*, segment tables. In all cases, views provide a mechanism for managing ephemeral state while providing enough context for threads to execute.

7.2.2 *Threads*

Twizzler provides a set of threading primitives for applications. Threads in Twizzler are always attached to a view and one or more security contexts. Threads may communicate with each other using shared memory and can signal each other with a system call. Since everything in Twizzler is an object, each thread has a state object associated with it. Signals can be raised assuming the raiser has appropriate permissions on the state object, and the state object contains information about the thread.

A key primitive in Twizzler is the `thread-sync` system call. This call operates similar to `futex(2)` on Linux, except that it supports waiting on and waking up a number of different words of memory simultaneously. Multi-word thread-sync is necessary to support `select(2)`-like or `poll(2)`-like operations in a system where all data access is done with memory semantics. Twizzler's standard library exposes an API for event handling that uses multi-word thread-sync, where objects may expose a set of "events" that can be triggered and waited for. This is used in numerous places to implement event handling for multiple communications streams implemented in objects.

7.2.3 *Program Instancing*

Programs can be loaded and run as ELF objects specifically linked for Twizzler. We provide a linker script that links program data into virtual addresses that correspond to the first few view slots in an address space. This way an ELF file can be loaded as a simple copy-from operation from the ELF object into several new objects (*e.g.*, a `text`, `rodata`, and `data` object). Program components like thread-local storage, stack, and heap can also be created out of objects. Finally, Twizzler supports a `fork`-like operation that copies a view object into a new view, remapping and applying copy-on-write copy-from operations as needed.

7.3 SECURITY

Twizzler's focus on memory-based objects requires that we design the security model around hardware-based enforcement, where the MMU checks each access. This design is *inevitable* in a data-centric OS, since the kernel is not involved in every memory access. The kernel merely specifies the access rights when mapping an object and then relies on the hardware to enforce those rights with a low overhead.

A key design choice we make is *late-binding on security*. Applications request access to an object with permissions that they desire; if they access the object in only allowed ways (*e.g.*, only reading a read-only object), no fault occurs. This is because when we map an object (via a view), the kernel is not immediately involved, and so cannot check access rights for a particular access at the time the mapping is setup. Performing an access rights check on time of first access does not make sense either, as it associates a specific access (that might be allowed) with a permissions error. For example, if a program reads object A, and that program is allowed to read A, it should be allowed to perform the read even if it requested read-write access to the object. This late-binding enables simpler programs that need not worry about elevating access rights through remapping data objects. Programs can make progress without knowing in advance the permissions of the objects they might access, thus enabling the reuse of the OS's access control mechanism in applications, as we saw in the previous chapter.

Threads run in a security context [12, 41, 79], which contains a list of access rights for objects and allows the kernel to determine the access rights of programs. Using these contexts, Twizzler is able to provide analogues to groups and owners in UNIX while providing more fine-grained access control if necessary. Unlike past exploration into security contexts, data-centric OSes offer an advantage in simplicity. A security context abstraction in a UNIX-like OS needs to maintain access rights to a set of fundamentally different things, such as paths, virtual memory locations, and system calls. Instead, Twizzler's security contexts specify access rights to an object via IDs instead of virtual addresses. This also makes security contexts persistent, allowing us to use them as the primary way we assign security roles to threads.

7.3.1 *Security Contexts and Page Tables*

Previously, we discussed how view objects allow applications to specify what objects they want mapped in, and with what protections. These are merely *requests*, however. Of course if the thread does not have the appropriate *permissions* the kernel will program the address translation hardware appropriately. This presents a problem: since threads can attach to a number of different security contexts, the number of different page-table structures that the kernel needs to manage grows quickly.

Twizzler uses Intel's Extended Page Table (EPT) technology[59], which is part of the virtualization extensions. The EPT allows a virtual address to be translated by two separate page tables, and is commonly used to virtualize the MMU in virtual machines. The first level, using normal MMU page tables, translates a virtual address to an object-logical address—typically with second-level address translation this is referred to as the "guest-physical" address—and the second-level translates this to a physical address. We discussed this two-level scheme previously in Chapter 5, but here we can use the functionality to gain security benefits.

[59] Other architectures (and AMD) have similar systems, but Twizzler does not support them yet.

Two-level address translation via the EPT enables Twizzler to split protection *requests* and access control permissions. At the top-level, Twizzler applies the requested protections from the view maps without restriction, and programs the EPT to enforce access control derived from the currently active security context. Splitting protection and access control is what allows applications to map objects in for whatever access mode they would like without having to worry about first checking permissions. Furthermore, by separating out the permissions enforcement from ephemeral state and location mapping, we reduce the number of page table structures the kernel needs to manage from $O(nm)$ to $O(n+m)$, where n is the number of views and m is the number of security contexts. Views and security contexts can also be switched out independently from each other, which more closely fits the semantics of Twizzler.

Two-level mapping also greatly simplifies the design of the kernel. Since mapping objects to physical memory is done at the second level, page eviction is easy—the kernel can simply modify the shared page tables stored per-object, which updates the translation for all views and contexts on the system after appropriate coherence. Moving objects between DRAM and NVM is made easier because objects reside in a given location within object-logical space regardless of where they are mapped in virtual memory, so the kernel does not need to maintain back pointers to update page table structures.

7.3.2 *Virtualization Hardware*

Twizzler's use of virtualization hardware for normal operation is a limitation of existing processors. Intel does not have a mechanism for using the EPT without switching on the entire virtualization system and running in VMX-non-root mode. In practice, the additional overhead from running with virtualization is negligible because we do not need all the protection of a traditional virtual machine and so we can switch much of it off. Because Twizzler's kernel is its own guest, we can avoid much of the overhead introduced by VM exits necessary in lower-trust VM models. For example, Twizzler's kernel is allowed to modify the EPT structures itself, despite being virtualized, and modern processors contain extensions that allow the guest to switch out EPTs itself and handle EPT faults without triggering a VM exit.

This pairs nicely with using the IOMMU[60] as well—since EPT structures on Intel can be reused in the IOMMU, we can apply security contexts to drivers as well, making driver code less of a special case. For example, Twizzler provides a driver model for userspace drivers that allows driver code to construct security contexts that explicitly map in only the necessary objects that a device might need to access (*e.g.* command queues, data objects, *etc.*). As hardware devices grow in complexity and increase their autonomy, treating them as additional computation resources and limiting access to objects through mechanisms already in-place for normal applications allows simpler programming of advanced hardware devices.

[60] As much as the IOMMU is able to "pair nicely" with anything.

7.4 POSIX COMPATIBILITY

Twizzler provides a compatibility framework for POSIX applications. C applications link to a patched version of `musl` [118], a C library written for Linux. We have modified `musl` to emit function calls to a library we wrote, called `twix`, instead of directly issuing system calls. The calls into `twix` are then handled by emulating their Linux system call behavior. For example, our implementation of `write(2)` allows us to create pseudo-terminal objects that are shared between applications and a terminal emulator that is hooked up to the serial port. Thus applications can call `printf` to get their messages out to wider society. Rust applications do not need as much of a compatibility layer, both because we can modify Rust's standard library more easily, and because Rust is not as tied to POSIX as C is.

7.4.1 *Porting to Twizzler*

Porting in Twizzler is straight-forward. We have a collection of tools that provide a framework for compiling software using the Twizzler toolchain against other ported software and libraries. Since we have chosen `musl` as our standard C library, many applications work already with minor changes. However, it is often the case that applications require some small tweaks to get running—for example, configuration paths—an experience common for anyone who has ported software to a new operating system[61].

[61] You can't see it, but I am raising my glass to you, those who have shared in this experience.

To date, we have ported a number of tools one would expect to find on a Unix system, such as `busybox` (providing numerous command-line utilities), `bash`, `vim`, `gcc`, `binutils`, and others. Many of these programs required little or no modification. Of course, this means that they do not gain some of the benefits Twizzler's model provides, since they still operate on persistent data with a POSIX model, however our goal in porting these tools was not to improve their performance, it was to provide a somewhat familiar environment for users.

Perfect emulation of a Linux kernel is a huge effort, and it is not the primary goal of our research. As a result, not all system calls are implemented and Linux features like `procfs` are lacking. This means that some programs may require features that are not yet implemented, and therefore require modifications to `twix` to run. However, as we continue to port software, `twix`'s coverage of Linux features grows, making future porting easier. We will continue to implement more support in `twix` for applications as needs arise. Note that many applications (even complex applications like `gcc`) often boil down to reading and writing files and managing processes, all of which is implemented.

7.4.2 *Twix System Call Overhead*

Our Unix emulation layer, `twix`, is meant to provide compatibility for legacy applications. While we expect that applications will wish to take full advantage of NVM and Twizzler's improvements in programmability and performance, we can still provide a small benefit for applications that rely on `twix` to provide POSIX-like I/O. Access to `twix` is done by `musl`, the C library we use, when it would normally perform a system call to a Linux kernel. We replaced all instances of the `syscall` instruction in C and assembly code in `musl` with a `call` instruction to an entry point in `twix`. This entry point, despite being a function call, obeys the Linux system call ABI (*e.g.* which registers hold parameters). Thus while it has significantly less overhead than a full system call and context switch, it does still have higher overhead than a normal function call since it must back up and restore all registers.

System Call	OS	Average Latency (ns)
`getpid`	Linux	98.7 ± 2.3
	Twizzler	10.2 ± 0.2
`read`	Linux	321.4 ± 0.2
	Twizzler	55.4 ± 0.2

TABLE 7.1

Latency of selected `twix` system calls compared to Linux system calls.

Table 7.1 shows the latency of some selected system calls on both Linux and Twizzler (implemented via `twix`). As expected, `getpid`'s overhead is small on both systems, but on Twizzler it is significantly lower. The difference, in this case, comes largely from the kernel entry overhead. A small amount of additional overhead comes from `twix` matching the Linux system call ABI and having to call its `getpid` implementation through a lookup table.

We also measured the latency of a call to `read` for a file. We chose to do reads on cached files for a small number of (already cached) bytes to avoid device transfer overhead. Performing a file read on Twizzler often amounts to a call to `memcpy`, so applications that perform large numbers of small reads could see some benefit. In contrast, on Linux, the kernel needs to traverse internal file structures, the page-cache, and possibly file system structures. However, as we said, `twix` is intended for legacy support, not performance, despite the lower system call overhead.

7.5 CONCLUSION

We now have an understanding of what addressing and pointers look like in Twizzler, and a basic idea of the operating system services Twizzler provides. One can write programs, multi-thread them, synchronize across threads via object sync words, access, create, copy, and delete objects, and use UNIX compatibility for things like `printf`. But this still doesn't address issues of memory safety, types, and failure-atomicity. Join us in the next chapter where we will start diving into these scary topics.

8 PROGRAMMING MODEL

SYNOPSIS *This chapter will discuss the programming model of Twizzler at a higher level than operating system interfaces, including object layout, safety, and lifetime.*

POKE: Fear of failure is a poor reason not to try.

—Outer Wilds

While the direct interfaces presented by the operating system are important, most programmers will not directly interact with them. Instead, they will use higher-level interfaces provided by a standard set of Twizzler libraries. We will now take a look at some of the aspects of writing programs that use objects on Twizzler. Note, though, that we are still leaving a lot up to specific language runtimes and instead of being doctrinaire about all aspects of the system, we prefer to allow flexibility. An example of this, which we will explore in more detail in this chapter, is not preventing "persistent to volatile" references at a system level, and instead relying on higher-level runtimes to enforce such things.

8.1 OBJECT LAYOUT

Recall that objects are, fundamentally, a "bag of bytes", all identified via an ID and an offset, with some areas pre-defined, for example, the FOT. Objects in Twizzler often have a header at the object's base, the contents of which depend on what the object contains. Often these headers have pointers to other data in the object, and describe the type of the object. For example, in our evaluation we implement a red-black tree in an object. The header contains some basic information about the tree as well as a pointer to the root node. Placing headers at the object's base gives applications a "starting point" that they can use to start accessing object data. Twizzler provides a dedicated function to get a pointer to an object's header, called `obj_base`.

Note that the base address of an object is *not* at offset 0, but instead one page up, so that we can still trap NULL pointers. If this were not the case, a pointer value of 0 would still be a valid pointer, and we want to remain backwards compatible with the assumption that a NULL pointer has integer value 0. The bottom page of an object is unmapped by Twizzler, allowing NULL pointer dereferences to be trapped by the kernel.

While objects are flat, contiguous regions of memory, different applications may want to organize that memory in different ways. Some objects, such as `views` are largely interpreted as an array, but sometimes

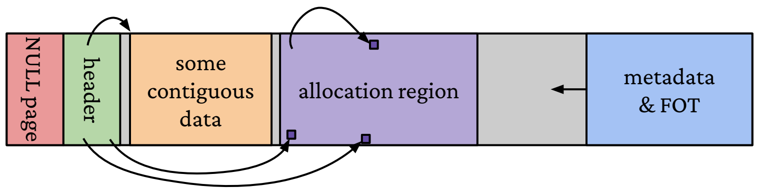

FIGURE 8.1

A typical object layout. The header, contiguous region, and allocation region are all optional, however most objects will have a header. This object contains a number of internal pointers between regions. The metadata region (which includes the FOT) grows downward.

applications need to explicitly allocate and deallocate memory within an object. Twizzler provides an API to allocate and free units of memory from application-specified regions within objects. We make use of this in our red-black tree code, where new nodes are allocated out of the object using this API.

Figure 8.1 shows a typical object in Twizzler. The NULL page is always present to trap NULL pointers, and is followed by a header. The application setting up this object may have a region of some contiguous data, such as some strings, or an array, and may point to it from the header. The object may have a region setup for allocation so that a future application using this object can easily allocate and free memory when manipulating the object. Finally, the FOT and metadata regions start at the top of the object and grow downwards.

ENSURING BASE TYPE PROPERTIES When accessing persistent or shared data for the first time, it's necessary to specify the type of the data. For objects, this means we need to specify the type of the header, as objects are typed by their header, also called the object "base". While in C the function that gives access to the base returns a `void *`, the function in Rust returns a reference to the specified base type. From there, any other data accessed traverses data structures that use the type system. Twizzler's invariant pointers' Rust implementation is typed, and the mutability rules ensure that our transaction engine properly ensures transactional properties, so any data to which we can get a reference is well-typed as long as the header of the object is well-typed.

To ensure this, we encode, in the object's metadata, a pair of 64 bit numbers that refer to a unique ID of the header type[62] and a version number. When code tries to access the header, we do a one-time check to see if the recorded ID and version match the ID and version from the type that we are supposed to return. If they fail to match, we can return an error, and if they do match, then we know it's the correct type (up to collisions in the ID space, that is).

[62] This is currently allocated manually, though we plan to automatically generate the IDs via procedural macros in the future.

8.2 THE FLIPSIDE

One aspect of programming for NVM that is somewhat orthogonal to Twizzler's primary object model is considering what effects writes will have to the actual physical media. This kind of consideration is common—we think about block layouts and overwrites when considering differences between magnetic disk and SSDs—but for NVM, we should ask what the new device characteristics are, since it is important that systems are optimized to leverage their strengths and avoid stressing their weaknesses. We typically try to reduce writes to physical media, however with NVM it turns out that the number of *bits flipped* may be the more important metric.

NVMs such as PCM suffer from both wear and energy use when it must flip a stored bit. Applications such as IoT devices cannot tolerate wear out and energy use as easily as other applications, and these devices make good targets for the already dense and power efficient NVM technologies. Flipping a bit in PCM cells consumes $15.7-22.5\times$ more power than reading a bit [36, 37, 75, 99], and causes wear (whereas reading causes comparatively little wear). Thus, controllers can optimize by examining bits to determine which must actually change on a write [129]. Of course, such an optimization is dependent on software, which issues the writes. Reducing bit flips, an optimization goal that has yet to be sufficiently explored, can both save energy and extend the life of NVM, but we can optimize better by designing software to reduce bit flips.

We examined a number of common data structures and strategies for how we can reduce bitflips caused by data structure updates. This involved taking previously-known tricks like XOR linked lists [114] and generalizing them to trees and hash tables, along with choices for bit packing and data layout, and modifying Gem5, a system simulator, to count bit flips. We found that we could easily reduce a significant number of bits flipped in the data structures we studied with little performance impact. Further details can be found in Appendix A.

8.3 CRASH CONSISTENCY

Twizzler provides primitives for building crash-consistent data structures. At a low level, it provides mechanisms for writing back cache-lines, appropriate fences, and basic transactions. Applications use these primitives today outside of Twizzler to build up larger, more complex support for crash-consistent data structures.

PERSISTENT MEMORY Programming against persistent memory has some advantages compared to programming against volatile memory and needing to flush dirty pages to stable storage. Not only is it much

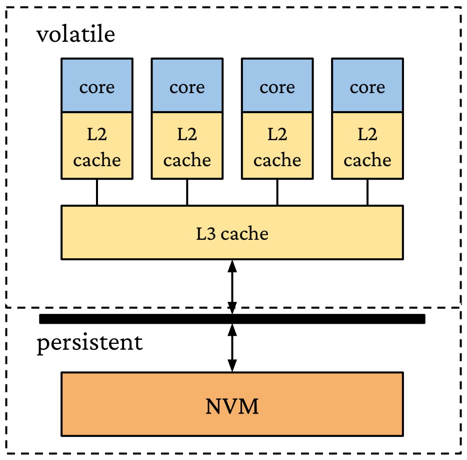

FIGURE 8.2
Volatile and persistent domains. Once the memory controller has acknowledged the write, the system guarantees persistence of the write.

faster, but failure-atomicity of an NVM data structure can be maintained by the application alone (if it so desires). Our goal is to provide low level primitives without restricting programs or prematurely prescribing particular solutions. There is a wealth of research on crash-consistent data structures for NVM [24, 38, 39, 81, 89–91, 93, 122], but it is still in flux. Of course, Twizzler manages *system* data structures, such as FOT entries, views, *etc.*, in a crash-consistent manner using the aforementioned primitives, locking, and fencing.

At the time of writing, Intel systems handle NVM by enabling a certain level of control over the cache system by applications, notably via the `clwb` and fence instructions. Figure 8.2 shows the boundary between persistent and volatile domains in an abstracted CPU that operates roughly like how Intel's do. In this model, any data stored in the cache is volatile, and will be lost if power is lost. Any data that makes it to the memory controller (for which the memory controller acknowledges the write) is persisted. A small amount of internal residual power is used by the persistent memory device to ensure the writes that reached the memory controller are stable.

With this model in mind, we can ensure failure atomicity in a rather inelegant manner if we use the `clwb` instruction to ensure cache lines are flushed and the appropriate fence instructions to ensure durability, and end up with a programming model reminiscent of multithreaded code using fences. We can abstract this a little bit to allow the flushing of *types* so as to avoid the programmer having to think entirely in cache lines. However, this still is a lower level than most programmers likely want for ensuring safety. While Twizzler does provide a set of interfaces for persisting data at this level, it also supports a basic transactional interface.

C TRANSACTIONS INTERFACE Twizzler provides a transactional-persistent logging mechanism. Programmers write TXSTART/TXEND[63] blocks to denote transactions and TXRECORD statements to record pre-changed values, similar to the mechanism provided by PMDK [102]. If applications need more complex transactions using different logging mechanisms, they can use libraries. Twizzler's internal data structures and libtwz's manipulation of object metadata is handled via a combination of these transactions and cache-line writebacks.

[63] If you "love" C macros, give these a look sometime.

RUST TRANSACTION INTERFACE The Rust interface to transactions is much safer than the C version, as it actually enforces various rules about type safety, aliasing, and mutability. An example transaction block is written like,

```
obj.tx(|tx| {
    let base = obj.base_mut(tx); // :&mut BaseType
    base.do_something();
});
```

Here we start a transaction and get access to a transaction handle (tx). When we want to get a mutable reference to part of the object, we must pass in a transaction handle, and the only way to get a transaction handle is via the tx function. Thus we can enforce the aliasing rules for Rust inside the transaction framework.

THREAD RESTART Twizzler provides a mechanism for restarting threads when power is restored following a crash. Since views are persistent objects, all objects mapped during a thread's execution are known across power cycles, and are mapped back in. The thread is then started at a special _resume entry point, allowing the program to handle the power failure in an application-specific manner. Of course, *volatile* objects will be lost when power resumes, and thus any attempted access to these objects will result in an exception. Applications that wish to resume after power failure will need to be aware of and handle this. We do not wish to prescribe any restrictions here—applications that want to place their heap in volatile memory for performance or security reasons should be allowed to. We expect higher level support for applications to manage persistent data, such as language support for persistent heaps, to make use of the features we provide, so applications that want to resume can put resuming information in persistent objects.

The reason we choose to restart threads at a known, different entry point from normal application start up is that in current systems, there is always volatile computation state (*e.g.* registers, the cache) that is lost when power is lost. Of course, in the future, systems may be able to prevent the loss of more and more ephemeral computation state (with the logical extreme being perfect resumability). In this case, the _resume handler can be a simple stub that resumes the execution exactly as left off. The more likely case, periodic checkpointing, can be similarly handled, with the `_resume` handler selecting the most recent valid check point to resume from. The `_resume` handler enables all of these solutions, thus remaining applicable across hardware evolution.

8.4 MEMORY SAFETY AND LIFETIMES

One final aspect of objects is safety, that is, ensuring proper typing of memory and ensuring that object memory does not leave a dangling reference behind. Twizzler accomplishes this via a combination of some runtime checks, some kernel support for specifying lifetime relationships between objects, and some language support.

8.4.1 *Object Types, Persistence, and Lifetime*

Applications need to be able to specify what *type* of memory an object resides in. Currently, we are operating on systems that contain both persistent NVM and volatile DRAM as main memory, and applications may want to make use of both of these memory types. Placing certain objects in DRAM, for example, can result in performance improvements (*e.g.* caching read-only objects) or security improvements (*e.g.* making temporary key material volatile). Twizzler exposes this choice to applications at object creation time, allowing them to specify the type of the object. At least two types, *volatile* and *persistent*, are supported by default. As additional types of physical memory are added to systems (*e.g.* different kinds of NVM with different properties, high-bandwidth memory, *etc*), applications may wish to have more fine-grained control over where objects are placed, and Twizzler's APIs allow such control. Objects can also be moved between types of memory after creation, though this may be a time consuming operation as it involves copying potentially large amounts of data[64].

[64] I would love to see NVM to/from DRAM direct physical copy hardware!

By default, objects are persistent and live in kernel-managed NVM or pager-managed storage unless they are marked as volatile. If an object is volatile, it has a limited lifetime that is related to the power state of the machine—as soon as power is lost or the system is rebooted all volatile objects disappear. Note that Twizzler removes the distinction between volatile and persistent objects for how applications *access* data, relying on higher-level language or library support and application support for dealing with the limited lifetime of volatile objects.

The property of persistent versus volatile for objects differs from the concept of ephemeral data. The "volatile" property places a physical restriction on the lifetime of an object (the machine's power state), while the “persistent” property indicates that the object will exist until explicitly deleted. Objects can also be long-lived or ephemeral independent of their persistence property, since we use the term “ephemeral” to describe information, data, or state that has a finite lifetime and is expected to “go away”. While all volatile objects are ephemeral, the reverse is not true—we may place ephemeral data in a persistent object to allow for recovery after an unexpected power cycle. The “persistent” property of an object is a recorded piece of information that the kernel associates with an object, but there is no such information for ephemeral versus long-lived. Instead, we provide a mechanism for specifying a logical lifetime of objects relative to one another with a mechanism called *ties*, which we will discuss below.

8.4.2 *Object Ties and Logical Lifetime*

Applications in Twizzler also have a lifetime; an application’s job is typically to operate on some persistent data while performing some computation before eventually exiting. Such an application will likely use volatile objects to represent temporary computation state (*e.g.* the stack and heap, which are ephemeral). However, just assigning an object as volatile is insufficient because there is a lifetime mismatch: the volatile object will live until the next reboot while the application may exit before then or may even live and try to recover after a power cycle. Manually deleting the volatile object when the application is done is also insufficient, as it does not account for crashes where the application may be unable to clean up its state. Furthermore, applications that wish to support recovery may make use of persistent stacks and heaps, thus these objects would have to be persistent despite being ephemeral.

While we could provide a mechanism designed specifically for this “system-level” task, where the kernel maintains a set of objects to automatically cleanup when an application exits, this would require the kernel to have some understanding of what an “application” is. Furthermore, if we generalize a solution to automatic cleanup, we can allow applications to make use of it for their own purposes. For example, in Unix, it is common for programs to create and immediately unlink files to ensure the system frees those resources when the program exits. We would like to reproduce similar semantics here that also solves the lower level problem above of freeing application state by assigning a lifetime to objects that is more expressive than simply “volatile” and “persistent”.

In Twizzler, object lifetime is expressed through *ties*. An object can be tied to another by invoking a system call that tells the kernel that object `A` is tied to object `B`, after which the lifetime of `A` is guaranteed to be at least that of `B`. The kernel will not fully delete object `A`, even if the delete system call is invoked on it, until after `B` is fully deleted. An object may be tied to a large (but finite) number of other objects and may also be *untied* at any time. This model of specifying object lifetime relative to others is similar to Rust [119], where reference lifetime can be named so that the programmer can express lifetimes of objects relative to each other. Note that object ties are not related to invariant pointers (discussed in more detail in Chapter 6), and instead primarily provide a way to formalize automatic cleanup.

Object ties provide a convenient mechanism for applications to build data structures across multiple objects without giving up easy cleanup if something goes wrong or if the "root" object is deleted. Twizzler also uses ties internally: when an object is created as copy-from an existing object, it uses copy-on-write semantics, and thus internally marks the source object as tied to the new object. We also tie ephemeral program state objects to threads (which are also represented by objects) such that they are automatically cleaned up when a program exits. It is our expectation that programmers will only rarely directly use ties. Instead, we expect that ties will provide necessary features that higher-level programming language support for persistent memory can use.

Note that object ties interact with the notion of volatile and persistent objects, because volatile objects have an implicit *maximum* lifetime—that of the next machine restart or power loss. Tying volatile objects to volatile objects and persistent objects to persistent objects both act as expected. Tying a persistent object to a volatile object is also semantically simple (persistent objects already have an "assumed lifetime" that is longer than a volatile object). Tying a volatile object to a persistent object, however, may seem somewhat nonsensical. However, Twizzler does still allow this because it has useful semantics: if an application creates a data structure with some volatile component[65], it may want to tie the lifetime of that volatile component to the persistent component if the data structure is to be deleted. This use case (creating a persistent object that we expect to delete) is not uncommon, particularly in applications designed to recover partial computation after a crash. Note that, in this case, the maximum lifetime of the volatile object is still in-play; after a power cycle, that object will no longer be present, so tying a volatile object to a persistent object is somewhat dangerous.

[65] Since Twizzler's kernel is not involved in reference creation, it cannot prevent such a reference from being created. We expect language support for persistent data structures to impose restrictions on applications in this regard, and the OS should not prematurely restrict how applications use volatile and persistent objects. Access to a volatile object that no longer exists after a reboot results in a simple access fault, mitigating security concerns.

8.4.3 *Safe Allocation*

As we discussed above, an object may optionally have a region within which we rely on an object memory allocator to organize memory. However, when objects may be persisted or shared, there are some additional hazards to consider.

1. **Crash Consistent**: The allocator must allow for power or application failures, and so all operations must be failure-atomic.

2. **Leaked Memory**: Say we allocate some memory from an object. Let's consider the interface described by `malloc` and `free`. We can imagine a situation where an application allocates some memory, but before we manage to store the pointer anywhere, the power is reset. Or, alternatively, after a node is removed from, say, a list, but before the call to `free` is made, the power dies. In both cases the memory is leaked, and cannot be recovered without some application-specific `fsck`-like check.

3. **Dangling Pointers**: Imagine that same scenario with freeing a node, but this time, the order of operations (remove node, then free) is not specified[66]. Or, similarly, we allocate some memory and write a new node, and link it, but the order of operations is wrong again. Both of these cases result in writing a pointer to memory considered freed by the allocator.

[66] This could happen to improper use of transactions or persist barriers.

One might wonder if there concerns are specific to NVM, where hardware persists behind a curtain, or more generally, perhaps when we are flushing periodically to disk. Certainly we must worry about the above issues if hardware is controlling persistence, but we still care even without NVM. The *semantics* of accessing an object are of sharing it or not (recall from Chapter 4 that persistence *is* sharing), and thus once it's shared it could be in multiple places in space and/or time. We must, therefore, make allocation and resource acquisition, and deallocation and reference removal failure-atomic. Consider a case where we interrupt an allocator operation after part of the allocator state is persisted to disk and the power fails. The problems above still exist, even for persistent objects stored on a disk or SSD.

To properly have an allocator that works on shared objects, we must build it to be *internally* crash consistent and present an external interface that allows applications to ensure *external* crash consistency. To do this, we will need to[67] change the interfaces to something richer than `malloc` and `free`:

[67] Thankfully!

```
int alloc(object *obj, size_t len,
    void **owner, uin64_t flags,
    void (*ctor)(void *mem, void *data), void *data);
int free(object *obj, void *p,
    void **owner, uint64_t flags);
```

The allocate function is built around an allocation and construction pattern that follows three steps:

1. Allocate memory from the allocator.

2. Construct the memory into a valid inhabitant of the intended type.

3. Record the reference to the new data into some data structure.

The freeing pattern occurs in two steps:

1. Remove the final (or only) pointer to the to-free memory by overwriting it (or clearing it to NULL).

2. Free the memory back to the allocator.

These steps are vital for persistence-safety, and it is vital that they occur in that order and atomically. Permuting the steps can allow for any of the problems we discussed above, and these steps are all needed to properly allocate an instance of a type and record a reference to it.

Internally, the allocator keeps an undo log for the operations performed by the allocator functions. When allocating, a region is selected and removed from internal data structures. The operations to remove the memory region from internal data structures are recorded to the log. The constructor function (the `ctor` argument) is called on the memory region. After the constructor function completes, the region is flushed to memory for persistence safety. Next, data pointed to by the `owner` argument pointer is recorded into the log, thus ensuring we cannot have dangling pointers. Finally, the value at `*owner` is written with the new invariant pointer to the allocated region, and the log is flushed.

The above allocation algorithm is safe because it ensures that, if the log is replayed, all internal data structures will be reset to contain the region, and the owner pointer is also reset to prevent the (now unallocated) region from being referenced. Since the region was not allocated, we can assume that none of that memory is referenced externally, and the contents do not matter[68]. Thus we can assume that region was filled with uninitialized memory, and needs not be restored[69]. This is not a mere optimization for restoring after a crash, nor is it an optimization for bulk flushes during allocations, it is vital for efficient implementation. The total amount of memory that needs to be flushed is $O(s)$, where s is the size being allocated, because we have to flush the entire memory region that holds the type for which we are allocating and the size choice is controlled by the caller. However, the number of items we record to the log is controlled by the allocator implementation, and can be made $O(1)$.

[68] For example, we can use that region for allocator-internal data structures!

[69] Any internal allocator data that needs to be restored inside the region will be properly reset by replaying the log.

For any internal operation, we count the number of items it has to add to the log, *e.g.*, if we remove a region from the free list, split it, and add one part to the free list, how many log records are created by those list operations. Since those are fixed operations, we can count the total number statically. If, however, an allocator custodial function runs an unspecified number of times, we cannot be sure ahead of time how many operations it will perform (*e.g.*, merging nodes in a list depends on the number of mergeable pairs, which depends on the length of the list). For these operations, we define a maximum allowed number of log entries, and break out of loops early if we run out. Thus we only need to count the number of operations *per iteration*, and then tweak the maximum log entries to allow most such loops to run for several, but a bounded number of, loops. As a result, we can set a maximum log length and allocate it statically within the region, instead of having to handle a variable and unbounded number of entries due to either internal or external causes.

8.5 CONCLUSION

Ensuring that programmers have access to a reasonable programming model that sits atop the operating system abstractions built for it is of utmost importance. Even if the lower levels of the system provides access to a fast, invariant reference implementation in a global address space that simplifies data models and transformations, programmers still need systems that make some of the harder parts easier. Twizzler provides a set of higher level APIs that facilitate day-to-day tasks like allocations and ensuring crash consistency.

Part III

EPILOGUE

SEPTIMUS: When we have found all the mysteries and lost all the meaning, we will be all alone, on an empty shore.
THOMASINA: Then we will dance. Is this a waltz?

—*Arcadia*, Tom Stoppard

9 CONCLUSION

RIEBECK: I learned a lot, by the end of everything. The past is past, now, but that's...you know, that's okay! It's never really gone completely. The future is always built on the past, even if we won't get to see it.

—*Outer Wilds*

Let's talk about this operating system we just designed and built.

9.1 TO LOOK AHEAD

"Where did you go to, if I may ask?" said Thorin to Gandalf as they rode along. "To look ahead," said he.

—*The Hobbit*, J.R.R Tolkien

We stand before a crossroads. Before us is a convergence of trends, a great upset within the memory hierarchy, faster and smarter networks and interconnects, the disaggregation of compute, and the ever increasing demands for concurrency and performance from software. The friction to reexamine our models and evolve our operating systems is low—provided we can produce a compelling reason for developers to invest in new ways of programming. Do we take the easy route, like so many times before, and try to shoehorn new technologies into existing interfaces, or do we take a step towards a model that expresses a data-centric view and better fits with hardware trends and the goals of software?

For us, the answer is clear[70].

[70] It's the second thing.

So, what happens next? There are two aspects of the future I want to discuss with respect to Twizzler—the future of the model we discussed and its resilience, and future research directions for Twizzler.

9.1.1 *Future Models*

Looking back, it really does appear, in hindsight, as if operating system interfaces plunged fully and purposefully into the endless maw of complexity. However, the decisions over the years were not made with the foresight required to predict what criticisms we today would aim at those choices. The model I have presented here is similarly a product of its time, made with some attempt at foresight but, ultimately, must be superceded when it, too, becomes too unwieldy to effectively use among new hardware and software characteristics. That said, I have made some attempts to design a general model that may be useful beyond *specific* hardware and instead is modeled after patterns:

1. Instead of focusing specifically on NVM DIMMs, we have designed our object model around the idea of a heterogeneous memory system in general. Should future kinds of memory be adopted, and mix-and-matched, or should NUMA become more common,

our model built for understanding data movement within physical memory as a first-class operation will be able to adapt.

2. Instead of trying to be doctrinaire at a low-level and limit our invariant references to the semantics of a single language, we provide a “lowest common denominator” in our implementation of references, and allow additional metadata to be stored in the FOT should richer semantics need additional storage. As a result, we can not only support languages of today, but also future languages and runtimes that may wish for a richer model.
3. The Twizzler kernel is small, largely because most functionality is pushed into userspace. Thus, not only can we more easily modularize, update, and replace components, but those components are closer to applications and can be communicated with more easily.
4. The design of our global address space is not limited to exist within a single machine, it is large enough that huge numbers of computers can interact with it wih low coordination. Should the size need to increase, we can do so with an application-agnostic procedure. Finally, the address space is usable both by multiple independent nodes, but also components within the nodes, enabling hardware devices to operate independently.

9.1.2 *Future Research*

I hope that we continue with exploration.

—Margaret H. Hamilton

A common saying asserts that a dissertation describes completed work, and thus any discussion of future work should reflect the nature that the work is completed. Yet I would be remiss if I didn’t adequately convey that the nature of an operating system does not include completeness, and for a research operating system, that means there are always future research directions we can explore. Here are a few.

COMPILER SUPPORT Twizzler’s clean-slate NVM abstraction reopens the possibility of co-evolving OSes, compilers and languages, and hardware. Standardized OS support for cross-object pointers provides a stationary target for both compilers and hardware to design towards, whereas application-specific solutions do not. Twizzler’s pointer translation functions are simple enough to be emitted by a compiler. We plan to explore adding basic compiler support for C and C++ to automatically interoperate with Twizzler so that persistent pointers are even more transparent to the compiler. Better still, we would like to study additional language-level support for persistent pointers, including type and lifetime annotations, such as the ones supported in Rust [119], for additional semantics the compiler can make use of when emitting code that operates directly on persistent data structures.

HARDWARE SUPPORT Hardware support, too, can be helpful in improving the performance of our pointer translations. With Twizzler providing a common framework, we can clearly state our needs to hardware. For example, hardware accelerated FOT access would improve the performance of pointer-heavy data structures. Segmentation support, allowing us to assign page-tables for each object and load them in as needed, would dramatically speedup memory mapping (and move memory management closer to the semantics of our programming model). Finally, first-class support for abstracting physical memory—a necessary feature for efficiently moving data around in a heterogeneous memory hierarchy in the face of numerous devices—would simplify the design of the kernel because we would not need to invoke the entirety of the virtualization hardware. We are interested in exploring modifications to RISC-V to better support Twizzler.

HIGHER LEVEL PROGRAMMING FRAMEWORKS The programming model we discussed in Chapter 8 provides a number of basic operations on objects, but one could imagine a higher level model. Implementing Rust APIs for more complex transactional semantics on shared objects would allow programmers to easily express atomic operations. A higher level framework that has awareness of application semantics via the FOT can perform optimizations on behalf of the application that would traditionally be difficult to implement, *e.g.* semantics-aware prefetching and caching in cooperation with the network, enabled by inspecting the FOT. A framework that also has knowledge of operations that an application may perform on an object could then perform different distribution and coherence strategies for CRDT [108] objects than for immutable or arbitrarily-mutable objects.

ALTERNATIVE STORAGE TECHNOLOGIES Twizzler meshes well with key-value SSDs, which extend the NVMe specification to include `put` and `get` operations. This would allow us to store and retrieve parts of objects based on their names rather than block addresses, thus greatly simplifying the storage system of the OS because it removes the need for a filesystem. Twizzler uses a userspace pager design for moving data between memory and indirectly-accessible storage; providing a more "native" interface for object-based storage will greatly improve the performance of this system.

9.2 LOOKING BEHIND

"And what brought you back in the nick of time?" "Looking behind," said he.

—*The Hobbit*, J.R.R Tolkien

Let's return to the principal hypothesis I put forward in Chapter 1.

> The data-centric model for designing operating system abstractions is not only viable, but demanded both by software and hardware trends.

In Chapters 2 and 3 we discussed the trends in hardware and software that led to the above statement, putting together the arguments that, (1) hardware is pushing us towards having in-memory data structures escape their traditional confinement, (2) software overheads induced by the context problem and operating system involvement are too high for modern hardware, (3) a data-centric model addresses the problems we have discussed via a method of progamming and storing data that requires no serialization and enables direct access and sharing. We also discussed why retrofitting is insufficient, covering Claim 1 from Chapter 1.

"The most important reason for going from one place to another is to see what's in between."

—*The Phantom Tollbooth*, Norton Juster

In Chapter 4 we discussed some details of the data-centric viewpoint, followed by elucidating Twizzler's place in the design space. Chapters 5 and 6 covered the global address space and invariant references, going into detail with case studies, performance analysis, and modeling to establish the viability of the model in terms of usability, and performance and space overhead, thus covering the remaining Chapter 1 Claims.

Chapters 7 and 8 wrap up by discussing operating system services and implementation, and programming model details. Discussing details like failure-atomicity in allocation, persistence, durability, and crash-recovery is vital, as it establishes where the complexity in the system still remains, despite many things getting easier.

9.3 FINAL REMARKS

SOLANUM: It's tempting to linger in this moment, while every possibility still exists. But unless they are collapsed by an observer, they will never be more than possibilities.

—*Outer Wilds*

Operating systems must evolve to support future trends in memory hierarchy organization. Failing to evolve will relegate new technology to outdated access models, preventing it from reaching its full potential, and making it difficult for OSes to evolve in the future. Twizzler shows a way forward: an OS designed around new hardware trends and software demands that provides new, efficient, and easy to use semantics for direct access to memory in a global address space. Twizzler will give us a system from which we can build a full NVM-based OS around a data-centric design and explore the future of applications, OSes, and processor design on a new memory hierarchy.

...How beautiful. It's different than I'd envisioned. Whatever happens next, I do not think it is to be feared.

—The Prisoner, *Outer Wilds*

Overall, we have shown that invariant pointers in Twizzler allow programmers to easily build composable and extensible applications with low overhead by removing the kernel from persistent data access paths, thereby improving the flexibility and performance. Our simpler programming model improved performance despite the small pointer translation overhead. Twizzler is easy to work with compared to existing systems, and we were able to both quickly prototype real applications with advanced access control features and port existing software, such as SQLite.

But those aren't the real lessons. Instead, consider how much of the operating system was under review after asking a simple question, "what if the lifetime of in-memory data extends beyond the ephemera?" This question was at the heart of this research, and led to all the work herein. It's not tied to a specific idea about persistent memory[71] or networking; instead, it's about programming, applications, and *data*. Reconsidering core ideas in programming and operating systems can lead to dramatic shifts in our understanding and designs. If we do not take the plunge by calling into question core beliefs and make moves towards building new things, we will be forever mired in the cage of our outdated systems. But there is hope—not from Twizzler alone, but from all the operating systems research that has exploded onto the scene.

[71] Though I won't deny that NVM kicked it off.

"It's a magical world, Hobbes, ol' buddy... Let's go exploring!"

—Calvin, *Calvin and Hobbes*, Bill Watterson

It's an exciting time, and I cannot wait for the next new operating system that asks, *what if things could be better*.

Part IV

APPENDIX

CHERT: We only get so much time, don't we? Ah, there was still more I wanted to do...How unlucky to have been born at the end of the universe.

—Outer Wilds

A RATHER FLIPPANT APPENDIX

A

SYNOPSIS *This appendix discusses some early work I did on bitflip optimization for non-volatile memories.*

A.1 INTRODUCTION

As byte-addressable non-volatile memories (NVMs) become more common [47, 60, 75], it is increasingly important that systems are optimized to leverage their strengths and avoid stressing their weaknesses. Historically, such optimizations have included reducing the number of writes performed, either by designing data structures that require fewer writes or by using hardware techniques such as caching to reduce writes. However, it is the number of *bits flipped* that matter most for NVMs such as phase-change memory (PCM), *not* the number of words written.

NVMs such as PCM suffer from two problems due to flipping bits: energy usage and cell wear-out. As these memory technologies are adopted into longer-term storage solutions and battery powered mobile and IoT devices, their costs become dominated by physical replacement from wear-out and energy use respectively, so increasing lifetime and dropping power consumption are vital optimizations for NVM. Flipping a bit in a PCM consumes $15.7-22.5\times$ more power than reading a bit or "writing" a bit that does not actually change [36, 37, 75, 99]. Thus, many controllers optimize by only flipping bits when the value being written to a cell differs from the old value [129]. While this approach saves some energy, it cannot eliminate flips required by software to update modified data. An equally important concern is that PCM has limited endurance: cells can only be written a limited number of times before they "wear out". Unlike flash, however, PCM cells are written individually, so it is possible (and even likely) that some cells will be written more than others during a given period because of imbalances in values written by software. Reducing bit flips, an optimization goal that has yet to be sufficiently explored, can thus both save energy and extend the life of NVM.

Previously, we showed that small changes in data structures can have large impacts in the bit flips required to complete a set of data structure modifications [13]. While it is possible to reduce bits flipped with changes to hardware, we can gain more by optimizing compiler constructs and choosing data structures to take advantage of semantic information that is not available at other layers of the stack; it is critical we design data structures with this in mind. Successful NVM-optimized systems need to target new optimizations for NVM, including bit flip reduction.

We implemented three such data structures and evaluated the impact on the number of writes and bit flips, demonstrating the effectiveness of designing data structures to minimize bit flips. These simple changes reduce bit flips by as much as $3.56\times$, and therefore will reduce power consumption and extend lifetime by a proportional amount, with no need to modify the hardware in any way. Our contributions are:

1. Implementation of bit flip counting in a full cycle-accurate simulation environment to study bit flip behavior.

2. Empirical evidence that reducing memory writes may not reduce bit flips proportionally.

3. Measurements of the number of bit flips required by operations such as memory allocation and stack frame use, and suggestions for reducing the bit flips they require.

4. Modification of three data structures (linked lists, hash tables, red-black trees) to reduce bit flips and evaluation of the effectiveness of the techniques.

This appendix is organized as follows. Section A.2 gives background demonstrating how bit flips impact power consumption and NVM lifetime. Section A.3 discusses some techniques for reducing bit flips in software, which are evaluated for bit flips (Section A.4) and performance (Section A.5). Section A.6 discusses the results, followed by comments on future work (Section A.7) and a conclusion (Section A.8).

A.2 NON-VOLATILE MEMORY AND BIT FLIPS

Non-volatile memory technologies [15] such as phase-change memory (PCM) [75], resistive RAM (RRAM, or memristors) [110, 117], Ferroelectric RAM (FeRAM) [47], and spin-torque transfer RAM (STT-RAM) [69], among others, have the potential to fundamentally change the design of devices, operating systems, and applications. Although these technologies are starting to make their way into consumer devices [60] and embedded systems [110], their full potential will be seen when they replace or coexist with DRAM as byte-addressable NVM. Such a memory hierarchy will allow the processor, and thus applications, to use load and store instructions to update persistent state, bypassing the high-latency I/O operations of the OS. However, power consumption, especially for write operations, and device lifetime are more serious concerns for these technologies than for existing memory technologies.

A.2.1 *Optimizing for Memory Technologies*

Data structures should be designed to exploit the advantages and mitigate the disadvantages of the technologies on which they are deployed. For example, data structures for disks are block-oriented and favor sequential access, while those designed for flash reduce writes, especially random writes, often by trading them for an increase in random reads [25]. Prior data structures and programming models for NVM [24, 39, 50, 85, 122, 127] have typically exploited its byte-addressability while mitigating the relatively slow access times of most NVM technologies. However, in the case of technologies such as PCM or RRAM, existing research ignores two critical characteristics: asymmetric read/write power and the ability to avoid rewriting individual bits that are unchanged by a write [15, 129].

For example, writes to PCM are done by melting a cell's worth of material with a relatively high current and cooling it at two different rates, leaving the material in either an amorphous or crystalline phase [100]. These two phases have different electrical resistance, each corresponding to a bit value of zero or one. The writing process takes much more energy than reading the phase of the cell, which is done by sensing the cell's resistance with a relatively low current. To save energy, the PCM controller can avoid writing to a cell during a write if it already contains the desired value [129], meaning that the major component of the power required by a write is proportional not to the number of bits (or words) *written*, but rather to the number of bits *actually flipped* by the write. Based on this observation, we should design data structures for NVM to minimize the number of bits flipped as the structures are modified and accessed rather than simply reducing the number of writes, as is more commonly done.

A.2.2 *Power Consumption of PCM and DRAM*

While our research applies to any NVM technology with expensive writes, we focus on PCM because its power consumption figures are more readily available. Figure A.1 shows the estimated power consumption of 1 GB of DRAM and PCM as a function of bits flipped per second, using power measurements from prior studies of memory systems [13, 21, 36, 37, 75, 99]. The number of writes to DRAM has little effect on overall power consumption since the *entire* DRAM must be periodically refreshed (read and rewritten); refresh dominates, resulting in a high power requirement regardless of the number of writes. In contrast, PCM requires no "maintenance" power, but needs a great deal more energy to write an individual bit (~50 pJ/b [8]) compared to the low overhead for writing a DRAM page (~1 pJ/b [75]). The result is that power use for DRAM is largely proportional to memory *size*, while power consumption for PCM is largely proportional to *cell change rate*. The exact position of the cross-

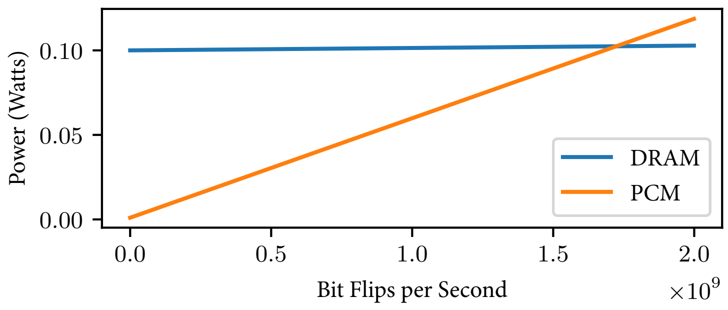

FIGURE A.1
Power use as a function of flips per second [13].

over point in Figure A.1 will be narrowed down as these devices become more common; many features of these devices, including asymmetric write-zero and write-one costs, increased density of PCM over DRAM, and decreasing feature sizes, will affect the trade-off point over time.

Figure A.1 demonstrates the need for data structures for PCM to minimize cell writes. Because the memory controller can minimize the cost of "writing" a memory cell with the same value it already contains, the primary concern for data structures in PCM is reducing the number of bit flips, which the memory controller cannot easily eliminate.

Power consumption is particularly concerning for battery-operated Internet of Things (IoT) devices, which may become a significant consumer of NVM technologies to facilitate fast power-up and reduce idle power consumption [63, 64]. Devices that collect large amounts of data and write frequently to NVM may find power usage increasing depending on access patterns. Thus, IoT devices may benefit significantly from bit-flip-aware systems and data structures.

A.2.3 *Wear-out*

Another significant advantage to avoiding bit flips is reducing memory cell wear-out. NVM technologies typically have a maximum number of lifetime writes, and fewer writes means a longer lifetime. However, by avoiding unnecessary overwrites, the controller would introduce uneven wear *within* NVM words where some of the bits flip more frequently than others due to biases of certain writes. For example, pointer overwrites may only alter the low-order bits, except for the few that are zero because of structure alignment in memory, if the pointers are to nearby regions. Thus, the middle bits in a 64-bit word may wear out faster than the lowest and highest bits. While reducing bit-flips increases the average lifetime of the cells in a word, it has the potential to exacerbate the uneven wear problem since such techniques might increase the biases of certain writes.

Fortunately, we can take advantage of existing research in wear-leveling for NVM that allows the controller to spread out the cell updates within a given word. While a full remapping layer similar to a flash translation layer is infeasible for NVM—the overhead would be too high—hardware techniques such as row shifting [130], content-aware bit shuffling [52], and start-gap wear leveling [98] may be able to mitigate biased write patterns with low overhead. This would allow NVM to leverage bit flip reduction to reduce wear even if the result is that some bits are flipped more frequently than others. These techniques, implemented at the memory controller level, can work in tandem with the techniques described in this appendix since they benefit bit flip reduction and can distribute "hot" bits across a word, mitigating the biased write patterns bit flip reduction techniques may introduce.

A.2.4 *Reducing Impact of Bit Flips in Non-volatile Memory*

Although bit flips in NVM have been studied previously, much of that work has focused on hardware encoding, which re-encodes cache lines to reduce bit flips, but re-encoding has limited efficacy [22, 62, 104] because it must also store information on *which* encoding was used. While hardware techniques are worth exploring, software techniques to reduce bit flips can be more effective because they can leverage semantic knowledge available in the software but not visible in the memory controller's limited view of single cache lines.

Chen *et al.* [21] evaluate data structures on NVM and argue that reducing bit flips is workload dependent and difficult to reason about, so we should strive to reduce writes because writes are approximately proportional to bit flips. We found that this is often *not* the case—our prior experiments revealed that bit flips were often *not* proportional to writes, and we were able to examine bit flips and optimize for them in an example data structure [13]. These findings are further corroborated by our experiments in Section A.4.

Since bit flips directly affect power consumption and wear, we can study three separate aspects for bit flip reduction:

1. **Data structure design**: Since data organization plays a large role in the writes that make it to memory, we designed new data structures built around the idea of *pointer distance* [114] instead of storing pointers directly. While *data* writes themselves significantly affect bit flips, these writes are often unavoidable (since the data must be written), while data *structure* writes are more easily optimized (as we see in existing NVM data structure research). Furthermore, data structures often require a significant number of updates over time, while data is often written once (since we can reduce writes by updating pointers instead of moving data). Thus the overall proportion of bit flips caused by data writes may drop over time as data structures are updated.

2. **Effects of program operation**: A common source of writes is the stack, where return addresses, saved registers, and register spills are written. Understanding how these writes affect bit flips plays a critical role in recommendations for bit flip reduction for system designers.

3. **Effects of caching layers**: Since writes must first go through the cache, it is vital to understand how different caching layers and cache sizes affect bit flips in memory. Complicating matters is the unique consistency challenges of NVM [24, 39, 122], wherein programs often flush cache-lines to main memory more frequently than they otherwise would, use write-through caching, or more complex, hardware-supported cache flushing protocols. These questions are evaluated in Section A.4.6.

A.3 REDUCING BIT FLIPS IN SOFTWARE

By reducing bit flips in software, we can effect improvements in NVM lifetime and power use without the need for hardware changes. To build data structures to reduce bit flips (Sections A.3.1–A.3.3), we propose several optimizations to pointer storage along with additional optimizations for indicating occupancy. For stack writes, we propose changes to compilers to spill registers such that they avoid writing different registers to the same place in the stack (Section A.3.4).

A.3.1 *XOR Linked Lists*

XOR linked lists [114] are a memory-efficient doubly-linked list design where, instead of storing a previous and next node pointer, each node stores only a `siblings` value that is the XOR between the previous and next node. If the previous node is at address p and the next node is at address n, the node stores $siblings = p \oplus n$. This scheme cuts the number of stored pointers per node in half while still allowing bidirectional traversal of the list—having pointers to two adjacent nodes is sufficient to traverse both directions. However, an XOR linked list has disadvantages; it does not allow $O(1)$ removal of a node with just a single pointer to that node, as a node's siblings cannot be determined from the node alone, and it increases code complexity by requiring XOR operations before pointers are dereferenced.

When they were proposed, XOR linked lists had little advantage over doubly linked lists beyond a modest memory saving. However, with the need for fewer bit flips on NVM, they gain a critical advantage: they cut the number of stored pointers in half, reducing writes, but they also store the XOR of two pointers, which are likely to contain similar higher-order bits, making the `siblings` pointer mostly zeros.

One problem with the original design for XOR linked lists is that each node stores $siblings = p \oplus n$, but for the first and last node, p or n are `NULL`, so the full pointer value for its adjacent node is stored in the head and tail. To further cut down on bit flips, we changed this design so that the head and tail XOR their adjacent nodes with themselves (if the node at address h is the head, then it stores $siblings = h \oplus n$ instead of $siblings = 0 \oplus n$). The optimization here is *not* a performance optimization—in fact, it's likely to reduce performance—and only makes sense in the context of bit flips, an optimization goal that would not be targeted before the introduction of NVM. However, with bit flips in-mind, it becomes critical. Other data structures may have similar optimizations that we can easily make to reduce bit flips [72].

[72] Circular linked lists solve the head and tail siblings pointer problem automatically, since no pointers are stored as `NULL`; however, in XOR linked lists this increases the number of pointer updates during an insert operation and requires storing two adjacent head nodes to traverse.

A.3.2 *XOR Hash Tables*

A direct application of XOR linked lists is chained hashing, a common technique for dealing with hash table collisions [28]. An array of linked list heads is maintained as the hash table, and when an item is inserted, it is appended to the list at the bucket that the item hashes to. To optimize for bit flips, we can store an XOR list instead of a normal list, but since bidirectional traversal is not needed in a hash table bucket, we need not complicate the implementation with a full XOR linked list. Instead, we apply the property of XOR lists that we find useful—XORing pointers.

Each pointer in each list node is XORed with the address of the node that contains that pointer. For example, a list node n whose next node is p will store $n \oplus p$ instead of p. In effect, this stores the distance between nodes rather than the absolute address of the next node and exploits locality in allocators. The end of the list is marked with a `NULL` pointer. In addition to a distance pointer, each node contains a key and a pointer to a value. The list head stored in the hash table is a full node, allowing access to the first entry in the list without needing to follow a pointer.

A second optimization we make is that an empty list can be marked in one of two ways: the least-significant bit (LSB) of the `next` pointer set to one, or the data pointer set to `NULL`. When we initialize the table, it is set to zero everywhere, so the data pointers are `NULL`. During delete, if the list becomes empty, the LSB of the next pointer in the list head is set to 1, a value it would never have when part of a list. This allows the data pointer to remain set to a value such that when it is later overwritten, fewer bits need to change. This is an example of an optimization that only makes sense in the context of bit flips, as it increases code complexity for no other gain.

A.3.3 *XOR Red-Black Trees*

Binary search trees are commonly used for data indexing, support range queries, and allow efficient lookup and modification, as long as they are balanced. Red-black trees [28, 103] are a common balanced binary tree data structure with strictly-bounded rebalancing operations during modification. A typical red-black tree (RBT) node contains pointers to its left child and right child, along with meta-data. They often also contain a pointer to the parent node, since this enables easier balancing implementation and more efficient range-query support without significantly affecting performance due to the increased memory usage [71].

We can generalize XOR linked lists to *XOR trees*. Instead of storing `left`, `right`, and `parent` pointers, each node stores `xleft` and `xright`, which are the XOR between each child and the parent addresses. This reduces the memory usage to the two-pointer case while maintaining the benefits of having a parent pointer, since given a node and one of its children (or its parent), we can traverse the entire tree. Like XOR linked lists, the root node stores `xleft = root ⊕ left`, where `root` is the address of the root node and `left` is the address of its left child, saving bit flips. To indicate that a node has no left or right child, it stores `NULL`.

Determining the child of a node requires both the node and its parent:

```
get_left_child(Node *node, Node *parent) {
  return (parent ⊕ node->xleft);
}
```

Getting a node's parent, however, requires additional work. Given a child c and a node n, getting n's parent requires we know *which* child (left or right) c is. Fortunately, in a binary search tree we store the key k of a node in each node, and the nodes are well-ordered by their k. Thus, getting the parent works as follows:

```
get_parent(Node *n, Node *c) {
  if(c->k < n->k) return (n->xleft ⊕ c);
  else return (n->xright ⊕ c);
}
```

Note that this is not the only way to disambiguate between pointers. In fact, it's not strictly necessary to do so because the algorithms can be implemented recursively without ever needing to traverse up the tree explicitly. However, providing upwards traversal can reduce the complexity of implementation and improve the performance of iteration over ranges. Another solution to getting the parent node would be to record whether a node is a left or right child by storing an extra bit along with the color. We did not evaluate this method, as it would increase both writes and bit flips over our method.

With these helper functions, we implemented both an XOR red black tree (`xrbt`) and a normal red-black tree (`rbt`) using similar algorithms.

The code for `xrbt` was just 20 lines longer, with only a minor increase in code complexity. Node size was smaller in `xrbt`, with a node being 40 bytes instead of 48 bytes as in `rbt`. To control for the effects of node size on performance and bit flips, we built a variant of `xrbt` with the same code but with a node size of 48 bytes (`xrbt-big`).

GENERALIZATION These techniques can generalize beyond a red-black tree. Any ordered k-ary tree can use XOR pointers in the same way. As discussed above, disambiguating between pointers during traversal depends on either additional bits being stored or using an ordering property. Either technique can work with arbitrary graph nodes.

A.3.4 *Stack Frames*

Data structure layout and data writes are only some of the writes made by a program. Register spills, callee-saved register saving, and return addresses pushed during function calls are all writes to memory, and if these writes make it to NVM, they will cause bit flips as well. These writes may make it to main memory if the cache is saturated or if the program is designed to keep program state in NVM to enable instantaneous restart after power cycles [89]. Additionally, systems designed for NVM may run with write-through caches to reduce consistency complexity, resulting in execution state reaching NVM.

The exact pattern of stack writes depends on the ABI and the calling convention of a system and processor, though we focus on x86-64 Linux systems. When a program calls a function, it (potentially) pushes a number of arguments to the stack, followed by a return address. In the called function, callee-saved registers are pushed to the stack, but *only* if they are modified during that function's execution. When finished, the callee pops all the saved registers and returns.

Our observation is that the order that callee-saved registers are pushed to the stack is *not specified*, meaning that two different functions could push the same registers in a different order. Secondly, the same callee-saved register is less-likely to change drastically in a small amount of code in a tight loop, since these registers are typically used for loop counters or bases for addressing. Thus, a loop that calls two functions alternately will likely have similar or the same values in the callee-saved registers during the invocation of both functions. If these two functions push the (often unchanged) callee-saved registers to the same place both times, fewer bit flips will occur than if the functions pushed them in different orders. While a simple example, such loops that call out to alternating functions with different characteristics can occur, for example, when rehashing a table, rebuilding a tree, or reading task items from a linked list.

We propose specifying a callee-saved register frame layout that functions adhere to, so that the registers are always pushed in the same order. To handle variable numbers of arguments, we make use of passing arguments in registers, common in many modern ABIs. If a function need not push any callee-saved registers, it can still reserve the stack space for that frame and then not push anything to save writes. Functions which only save a small number of registers can still push them to the correct locations within the frame. Finally, if this is standardized, programs need not worry about library calls increasing bit flips.

For example, if we have two functions `A` and `B` in an ABI where registers e, f, g, h are callee-saved, and `A` uses e while `B` uses g, then traditionally each function would simply push the frame pointer followed by the register they wish to save:

```
A:                      B:
push fp                 push fp
mov fp ← sp             mov fp ← sp
push e                  push g
...                     ...
pop e                   pop g
pop fp; ret             pop fp; ret
```

If e and g are significantly different, then a significant amount of needless bit flips could occur if these functions are called often. Instead, if we define a layout that functions adhere to for register saving, the code would look like:

```
                        B:
A:                      push fp
push fp                 mov fp ← sp
mov fp ← sp             sub sp, 16
push e                  push g
sub sp, 24              sub sp, 8
...                     ...
add sp, 24              add sp, 8
pop e                   pop g
pop fp; ret             add sp, 16
                        pop fp; ret
```

Here the code always pushes the same register to the same place, regardless of the registers it needs to save, thereby allowing overwrites by likely similar values. While it does add some additional instructions, code could instead write registers directly to the stack locations using offset style addressing, reducing code size.

A.4 MEMORY CHARACTERISTICS RESULTS

We evaluated XOR linked lists, XOR hash tables, and XOR red-black trees, tracking bits flipped in memory, bytes written to memory, and bytes read from memory during program execution. Our goal was not only to demonstrate that our bit flip optimizations were effective, but to also understand how different system and program components affected bit flips. In addition to tracking bit flips caused by our data structures, we also studied bit flips caused by varying levels and sizes of caching, calls to `malloc`, and writes to the stack. Finally, we evaluated the accuracy of in-code instrumentation for bit flips, which would allow programmers to more easily optimize for bit flips at lower cost than full-system simulation. All of these experiments were designed to demonstrate how effective certain bit flipping reduction techniques are. Existing systems are poorly equipped to handle evaluation of these techniques, since existing systems are poorly optimized for NVM. The techniques we present here are designed to be used by system designers when building *new*, NVM-optimized systems.

A.4.1 *Experimental Methods*

Evaluating bit flips during data structure operations requires more than simply counting the bits flipped in each write in the code. Compiler optimizations, store-ordering, and the cache hierarchy can all conspire to change the order and frequency of writes to main memory, potentially causing a manual count of bit flips in the code to deviate from the bits flipped by writes that *actually* make it to memory. To record better metrics than in-code instrumentation, we ran our test programs on a modified version of Gem5 [11], a full-system simulator that accurately tracks writes through the cache hierarchy and memory. We modified the simulator's memory system so that, for each cache-line written, it could compute the Hamming distance between the existing data and the incoming write, thereby counting the bit flips caused by each write to memory. The bit flips for each write were added to a global count, which was reported after the program terminated, along with the number of bytes written to and read from memory. This gave us a more accurate picture of the bit flips caused by our programs, since writes that stay within the simulated cache hierarchy do not contribute to the global count. We ran the simulator in *system-call emulation* mode, which runs a cycle-accurate simulation, emulating system calls to provide a Linux-like environment, while tracking statistics about the program, including the memory events we recorded.

We used the default cache hierarchy (shown in Table A.1) provided by Gem5, using the command-line options "`--caches --l2cache`". For

Cache	Count	Size	Associativity
L1d	1	64KB	2-way
L1i	1	32KB	2-way
L2	1	2MB	8-way

TABLE A.1
Cache parameters used in Gem5.

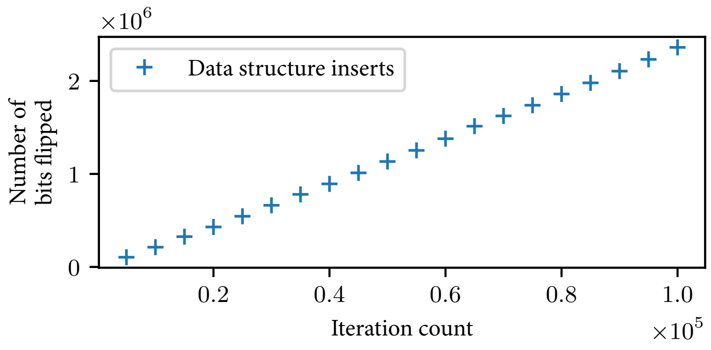

FIGURE A.2
A typical result of running a test program with increasing values of `iteration_count`.

the XOR linked list and stack writes experiments, we used `clwb` instructions to simulate consistency points (in the linked list, `clwb` was issued to persist the contents of a node before persisting the pointers to the node, and for stack writes, `clwb` was issued after each write). This was not done for the `malloc` experiment (we used an unmodified system `malloc` for testing), the XOR hash table (the randomness of access to the table quickly saturated the caches anyway), or manual instrumentation (caches were irrelevant). For the XOR red-black tree, in addition to the bit flip characteristics, we focused on observing how cache behavior affected more complex data structures; these results, along with the results of varying L2 size, are discussed in Section A.4.6.

Most of the test programs take an `iteration_count` as their first argument, which specifies how many iterations the program should run. For example, the red-black tree would do `iteration_count` number of insertions. We ran the simulator on a range of `iteration_counts`, recording the bits flipped, bytes written, and bytes read (collectively referred to as *memory events*) for each value of `iteration_count`. An example of a typical result is shown in Figure A.2. The result was often linear, allowing us to calculate a linear regression using gnuplot, giving us both a slope and confidence intervals. The slope of the line is "bit flips per operation"—for example, a slope of 10 for linked list insert means that it flipped 10 bits on average during insert operations. Throughout our results, only the slope is presented unless the raw data is non-linear. Since the slope encodes the bit flips per operation, we can directly compare variants of a data structure by comparing their slopes. Error bars are 95% confidence intervals.

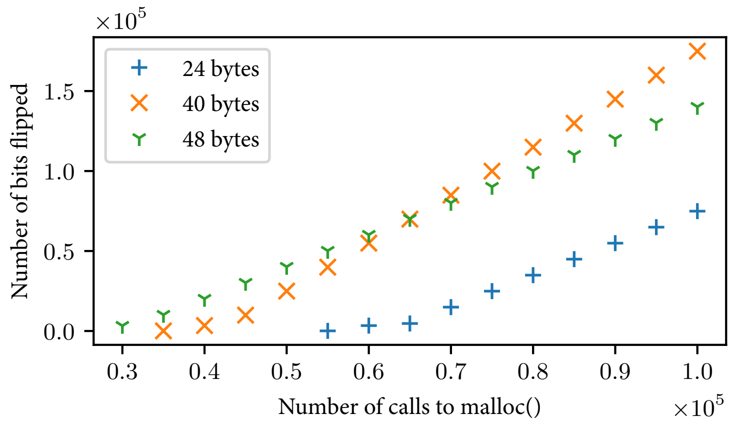

FIGURE A.3

Bit flips due to calls to `malloc`. Allocation size of 16 bytes is not shown because it matches with 24 bytes.

A.4.2 *Calls to `malloc`*

Many data structures allocate data during their operation. For example, a binary tree may allocate space for a node during insert or a hash table might decide to resize its table. An allocator allocating data from NVM must store the allocation metadata within NVM as well, so the internal allocator structures affect bit flips for data structures which allocate memory. Additionally, the pointers returned *themselves* contribute to the bits flipped as they are written.

We called `malloc` 100,000 times with allocation sizes of 16, 24, 40, and 48 bytes. We chose these sizes because our data structure nodes were all one of these sizes. The number of bits flipped per `malloc` call is shown in Figure A.3. As expected, larger allocation sizes flip more bits, since the allocator meta-data and the allocated regions span additional cache lines. Interestingly, 40 byte allocations and 48 byte allocations switch places partway through, with 40 byte allocations initially causing fewer bit flips and later causing more after a cross-over point. We believe this is due to 40 byte allocations using fewer cache lines, but 48 byte allocations having better alignment.

After a warm-up period where the cache hierarchy has a greater effect, the trends become linear, allowing us to calculate the bit flips per `malloc` call. Allocating 40 bytes costs $1.5\times$ more bit flips on average than allocating 48 bytes. Allocating 24 or 16 bytes has the same flips per `malloc` as 48 bytes but has a longer warm-up period, such that programs would need to call `malloc` (24) $1.56\times$ as often to flip the same number of bits as `malloc` (48).

While the relative savings for bit flips between `malloc` sizes are significant, their absolute values must be taken into consideration. Calls to `malloc` for 16 and 48 bytes cost 2 ± 0.1 flips per `malloc` (after the warm-up period) while calls to `malloc` for 40 bytes cost 3 ± 0.1 flips per

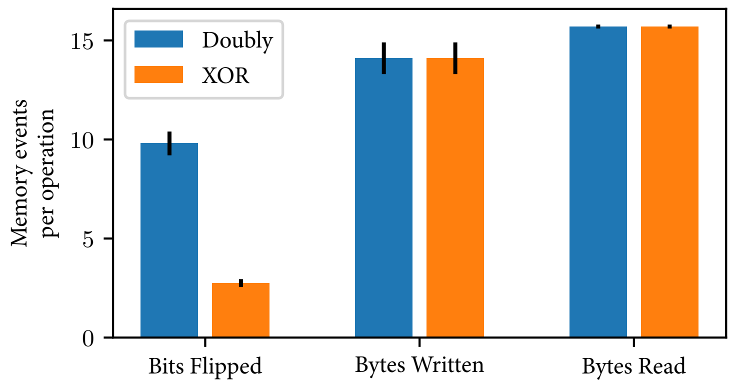

FIGURE A.4
Memory characteristics of of XOR linked lists compared to Doubly-Linked Lists.

`malloc`. As we will see shortly, the data structures we are evaluating flip tens of bits per operation, indicating that savings from `malloc` sizes are less significant than the specific optimizations they employ.

A.4.3 *XOR Linked Lists*

We evaluated the bit flip characteristics of an XOR linked list compared to a doubly-linked list, where we randomly inserted (at the head) and popped nodes from the tail at a ratio of 5:1 inserts to pops. The results are shown in Figure A.4. As expected, bit flips are significantly reduced when using XOR linked lists, by a factor of $3.56\times$. However, both the number of bytes written to and read from memory were the same between both lists. The reason is that, although an XOR list node is smaller, `malloc` actually allocates the same amount of memory for both.

We counted the number of pointer read and write operations in the code, and discovered that, although the XOR linked list performs fewer write operations during updates, it performs *more* read operations than the doubly-linked list. This is because updating the data structure requires more information than in a doubly-linked list. However, Figure A.4 shows that the number of reads *from memory* are the same, indicating that the additional reads are always in-cache.

A.4.4 *XOR Hash Tables*

We implemented two variants of hash tables: "single-linked", which implemented chaining using a standard linked list, and "XOR Node", which XORs each pointer in the chain with the address of the node containing the pointer. We ran a Zipfian workload on them [14], where 80% of updates happen to 20% of keys[73], where keys and values were themselves Zipfian. During each iteration, if a key was present, it was deleted, while if it was not present, it was inserted. This resulted in a workload where

[73] Skew of 1, with a population of 100,000.

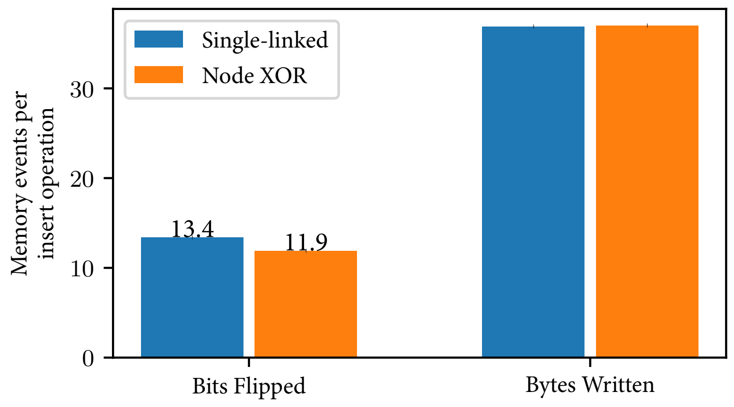

FIGURE A.5
Memory characteristics of XOR hash table variants under Zipfian workload.

a large number of keys were rarely modified, but a smaller percentage were repeatedly inserted or removed from the hash table.

Figure A.5 shows the bits flipped and bytes written by the hash table after 100,000 updates. As expected, the XOR lists saw a reduction in bit flips over the standard, singly-linked list implementation while the number of bytes written were unchanged. We were initially surprised by the relatively low reduction in bit flips ($1.13\times$) considering the relative success of XOR linked lists; however, the common case for hash tables is short chains. We observed that longer chains improve the bit flips savings, but forcing long hash chains is an unrealistic evaluation. Since buckets typically have one element in them, and that element is stored in the table itself, there are few pointers to XOR, meaning the reduction is primarily from indicating a list is valid via the least-significant bit of the next pointer. The bit flips in all variants come primarily from writing the key and value, which comprise 9.3 bit flips per iteration on average. Thus, this data structure had little room for optimization, and the improvements we made were relatively minor—although they still translate directly to power saving and less wear, and are easy to achieve while not affecting code complexity significantly.

A.4.5 *XOR Red-Black Trees*

Figure A.6 shows the memory event characteristics of `xrbt` (our XOR RBT with two pointers, `xleft` and `xright`), `xrbt-big` (our XOR RBT with each node inflated to the size of our normal RBT nodes), and `rbt` (our standard RBT) under sequential and random inserts of one million unique items. Each item comprises an integer key from 0 to one million and a random value. Both `xrbt` and `xrbt-big` cut bit flips by $1.92\times$ (nearly in half) in the case of sequential inserts and by $1.47\times$ in the case of random inserts, a dramatic improvement for a simple implementation

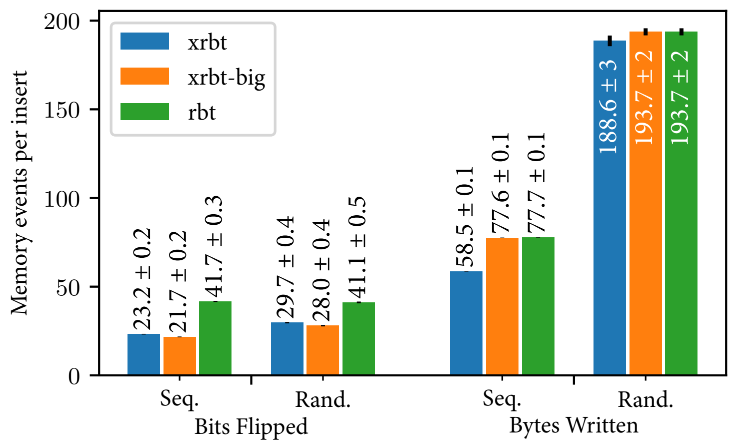

FIGURE A.6

Memory characteristics of XOR red-black trees compared to normal red-black trees.

change. The small saving in bit flips in `xrbt-big` over `xrbt` is likely due to the allocation size difference as discussed in Section A.4.2.

The number of bytes written is also shown in Figure A.6. Due to the cache absorbing writes, `xrbt-big` and `rbt` write the same number of bytes to memory in all cases, even though `rbt` writes more pointers during its operation. We can also see a case where the number of writes was not correlated with the number of bits flipped, since `xrbt` writes fewer bytes but flips more bits than `xrbt-big`.

We did not implement and test delete operation in our red-black trees because the algorithm is similar to insert in that its balancing algorithm is tail-recursive and merely recolors or rotates the tree a bounded number of times. Since the necessary functions to implement this algorithm are present in all variations, it is certainly possible to implement, and we expect the results to be similar between them.

A.4.6 *Cache Effects*

Although it is easy to exceed the size of the L1 cache during normal operation of large data structures at scale, larger caches may have more of an effect on the frequency of writes to memory. Of course, a persistent data structure which issues cache-line writebacks or uses write-through caching bypasses this by causing *all* writes to go to memory[74], but it is still worth studying the effects of larger write-back caches on bit flips. They may absorb specific writes that have higher than average flips, or they may cause coalescing even for persistent data structures worrying about consistency.

[74] Even if this is the case, a full system simulator will give a more accurate picture than manually counting writes, since store ordering and compiler optimizations still affect memory behavior.

We studied cache effects in two ways—how the mere presence of a layer-2 cache affects the data structures we studied and how varying the size of that cache affects them. Figure A.7 shows `xrbt` compared with

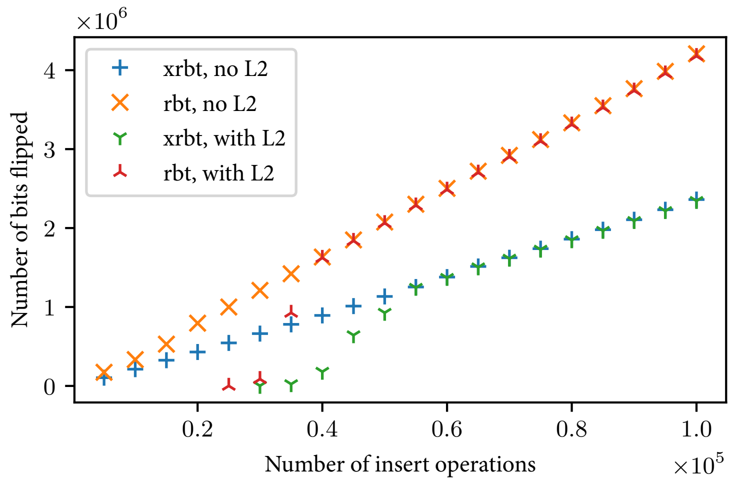

FIGURE A.7

Bits flipped by XOR red-black trees and standard red-black trees over a varying number of sequential inserts, with and without the L2 cache present.

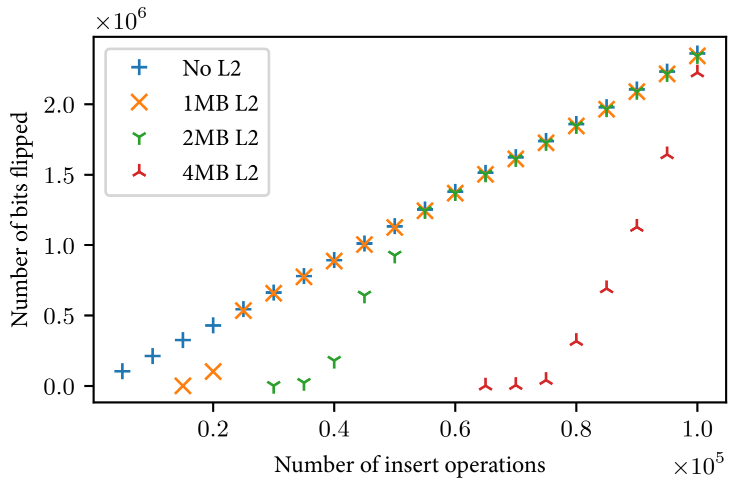

FIGURE A.8

Bits flipped by XOR red-black tree over a varying number of sequential inserts, with different sizes for L2.

`rbt`, with and without L2. The effect of L2 is limited as the operations scale, with the bit flips for both data structures reaching a steady, linear increase once L2 is saturated. The bit flips per operation for both data structures with L2 is the same as without L2 once the saturation point is reached, indicating that while the presence of the cache *delays* bit flips from reaching memory, it does little to reduce them in the long term. Finally, since `xrbt` has fewer bit flips overall and fewer memory writes, it took longer to saturate L2, delaying the effect.

Next, we looked at different L2 sizes, running `xrbt` with no L2, 1MB L2, 2MB L2 (the default), and 4MB L2, as shown in Figure A.8. The exact same pattern emerges for each size, delayed by an amount proportional to the cache size. This is to be expected, and it further corroborates our claim that cache size has only short-term effects.

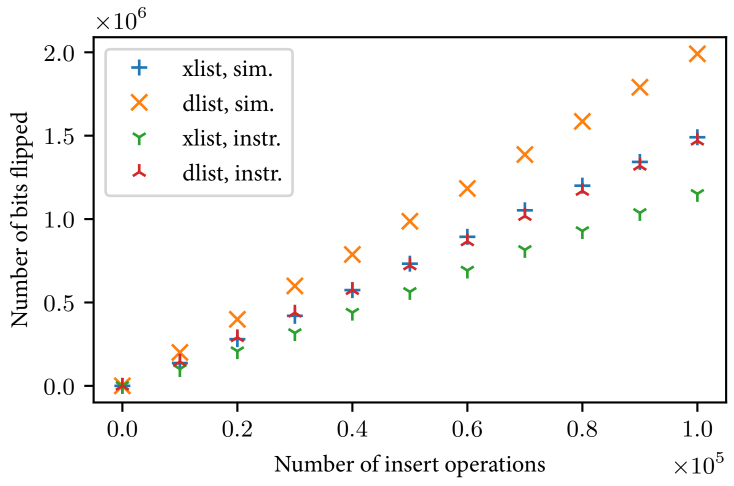

FIGURE A.9
Manual instrumentation for counting bit flips (instr) compared to full-system simulation (sim).

A.4.7 *Manual Instrumentation*

While testing data structures on Gem5 was straightforward, if time consuming, more complex structures and programs may be difficult to evaluate, either due to Gem5's relatively limited system call support or due to the extreme slowdown caused by the simulation. Since real hardware does not provide bit flip counting methods, we are left with using in-program instrumentation if we want to avoid the Gem5 overhead. However, these results may be less accurate.

To study accuracy of in-code instrumentation, we manually counted bit flips in the XOR and doubly-linked lists. We did this by replacing all direct data structure writes (*e.g.*, `node->prev = pnode`) with a macro that both did that write and also counted the number of bytes (by looking at the types), and computing the Hamming distance between the original and new values. Totals of each were kept track of and reported at the end of execution. While not difficult to implement, manual instrumentation adds the possibility of error and increases implementation complexity.

Figure A.9 shows the results of manual instrumentation compared to results from Gem5. While accuracy suffered, manual counting was not off by orders of magnitude. It properly represented the relationship between XOR linked lists and doubly-linked lists in terms of bit flips, and it was off by a constant factor across the test. We hypothesize that the discrepancy arose from the fact that our additional flip counting code affected the write combining and (possibly) the cache utilization. We expect that future system designs could "calibrate" manual instrumentation by running a smaller version of their system on Gem5 to calculate the discrepancy between its counts and theirs, allowing them to more accurately extrapolate the bits flipped in their system using instrumentation. Additionally, one could modify toolchains and debugging tools to

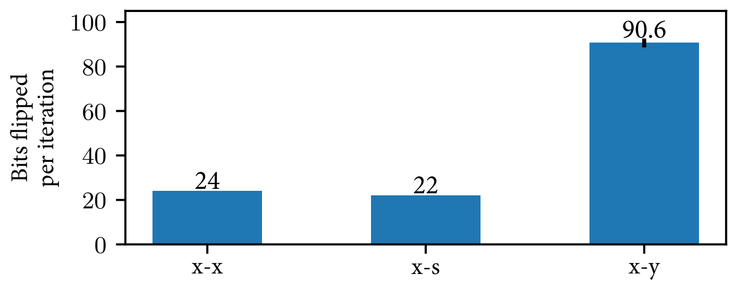

FIGURE A.10

Bits flipped by different stack frame layouts.

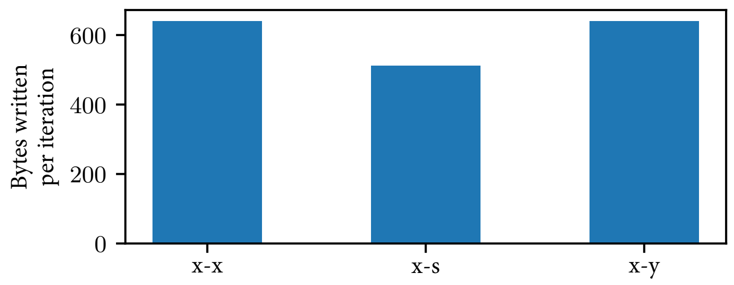

FIGURE A.11

Bytes written by different stack frame layouts.

automatically emit such instrumentation code during code generation. Manual instrumentation may find its use here for large systems that are too complex or unwieldy to run on Gem5, or as a way to quickly prototype bit flipping optimizations.

A.4.8 *Stack Frames*

To study bit flips caused by stack writes, we wrote an assembly program that alternates between two function calls in a tight loop while incrementing several callee-saved registers on x86-64. The loop could call two of three functions—function x, which pushed six registers (the callee-save registers on x86_64, including the base pointer) in a given order, y, which pushed the registers in a different, given order, and s, which pushed only two of the registers, but pushed them to the same locations as function x. Our program had three variations: **x-x**, which called function x twice, **x-s**, which alternated between functions x and s, and **x-y**, which alternated between functions x and y. The x-y variant represents the worst-case scenario for register spilling, while x-s demonstrates our suggestion for reducing bit flips. To force the writes to memory, we used `clwb` after the writes to simulate write-through caching or resumable programs.

Figure A.10 shows bit flips and Figure A.11 shows bytes written by all three variants. The x-s and x-x variants have similar behavior in terms of bit flips, which is understandable because they are pushing registers to the same locations within the frame. The x-y variant, however, had

Operation	XOR Linked	Doubly-Linked
Insert (ns)	45 ± 1	45 ± 1
Pop (ns)	27 ± 1	28 ± 1
Traverse (ns/node)	2.6 ± 0.1	2.2 ± 0.1

TABLE A.2
Performance of XOR linked lists compared with doubly-linked lists.

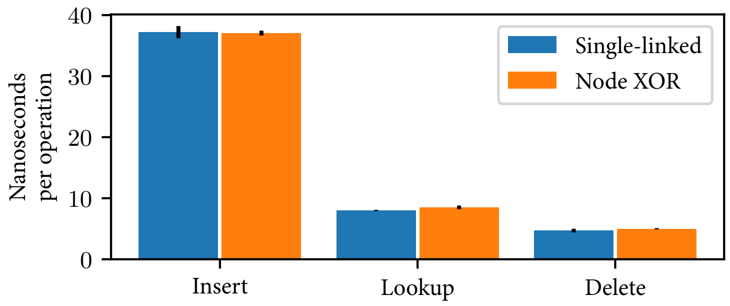

FIGURE A.12
Performance of XOR hash table variants.

$3.8\times$ the number of bit flips compared to x-x and $4.1\times$ the number of bit flips compared to x-s, showing that consistency of frame layout has dramatic impact. Unsurprisingly, x-x and x-y had the same number of bytes written, since they write the same number of registers, while x-s wrote fewer registers. By keeping frame layout consistent, we can reduce bit flips, and the optimization of only pushing the registers needed but to the same locations can further reduce writes as well.

A.5 PERFORMANCE ANALYSIS

While bit flip optimization is important, it is less attractive if it produces a large performance cost. We compared our data structures' performance to equivalent "normal" versions not designed to reduce bit flips. Benchmarks were run on an i7-6700K Intel processor at 4GHz, running Linux 4.18, `glibc` 2.28. They were compiled using `gcc` 8.2.1 and linked with GNU `ld` 2.31.1. Unless otherwise stated, programs were linked dynamically and compiled with `O3` optimizations.

XOR LINKED LISTS The original publication of XOR linked lists found little performance difference between them and normal linked lists [114]; we see the same relationship in our implementation (see Table A.2). The only statistically significant difference was seen in traversal, where XOR linked lists have a $1.18\times$ increase in latency; however, both lists average less than three nanosecond-per-node during traversal.

XOR HASH TABLES Figure A.12 shows the performance of the two hash table variants we developed. We inserted 100,000 keys, followed

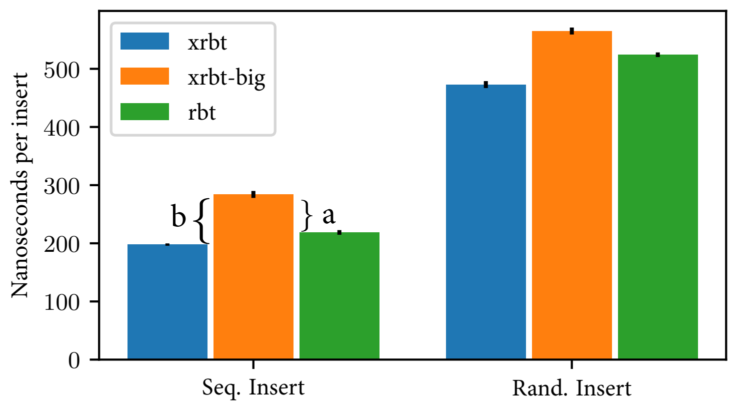

FIGURE A.13
Insert latency for XOR red-black trees compared to normal red-black trees. The label “a” shows the cost of the XORs, while “b” shows the cost of the larger node.

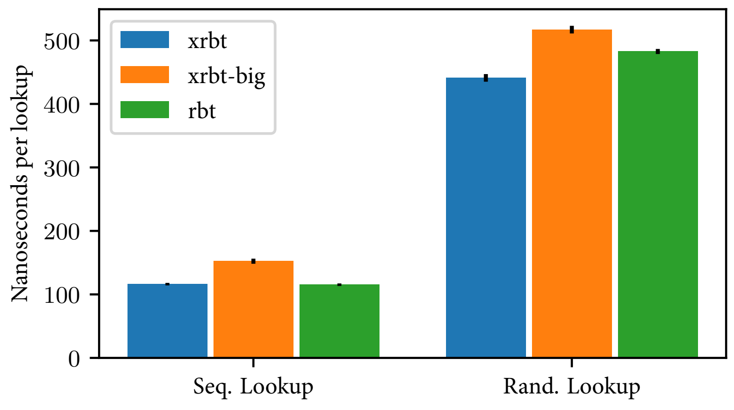

FIGURE A.14
Lookup latency for XOR red-black trees compared to normal red-black trees.

by lookup and delete. As expected, both variants have similar latencies, with a slowdown of only $1.06\times$ for using XOR lists during lookup.

XOR RED-BLACK TREES We measured `xrbt`, `xrbt-big`, and `rbt` during 100,000 inserts and lookups, the results of which are shown in Figure A.13 and Figure A.14. During insert, `xrbt` is slightly faster than `rbt`, with `xrbt-big` being slower, indicating that although there is a non-zero cost for the XOR operations, it is outweighed by the performance improvement from smaller node size and better cache utilization. The lookup performance shown in Figure A.14 demonstrates a similar pattern, although for sequential lookup the overheads are similar enough that there is no significant performance difference between `xrbt` and `rbt`.

A.6 DISCUSSION

SOFTWARE BIT FLIP REDUCTION The data structures presented here are both old and new ideas. While not algorithmically different from existing implementations (both `xrbt` and `rbt` use the same, standard red-black tree algorithms), they present a new approach to implementation with optimizations for bit flipping. This has not been sufficiently studied before in the context of software optimization; after all, there is no theoretical advance nor is there an overwhelming practical advantage to these data structures outside of the bit flip reduction, an optimization goal that is new with NVM. However, keeping this in mind has huge ramifications for data structures in persistent memory and applications for new storage technologies, as it presents a whole new field of study in optimization and practical data structure design. The goal is not performance improvements; instead we strive to prolong the lifetime of expensive memory devices while reducing power use, with at most a minor performance cost. These improvements can be achieved without hardware changes, meaning even savings of 10% ($1.1\times$) or less are worthwhile to implement because savings are cumulative.

These optimizations are not specific to PCM; any memory with a significant read/write disparity and bit-level updates could benefit from this. The energy savings from bit flip optimization will, of course, be technology-dependent, numbers for which will solidify as the technologies are adopted. Our estimates of the linear relationship between flips and power use (Figure A.1) indicate that, on PCM, the energy savings will be roughly proportional to the bit flip savings since the difference between read and write energy is so high.

Bit flips can and should be reasoned about directly. Not only is it possible to do so, but the methods presented here are straightforward once this goal is in mind. Furthermore, while reducing writes *can* reduce bit flips, we have confirmed that this is not *always* true. XOR linked lists reduced bit flips without affecting writes, while `xrbt` reduced writes over `xrbt-big` at the cost of increasing bit flips. With stack frames, the biggest reduction in bit flips corresponded with no change in writes, while the reduction of writes was correlated with only a modest bit flip reduction.

The implications are far-reaching, especially when considering novel computation models that include storing program state in NVM. Writes to the stack also affect bit flips, but these can be dramatically optimized. Compilers can implement standardized stack frame layouts for register spills that save many bit flips while remaining backwards compatible since nothing in these optimizations breaks existing ABIs. Further research is required to better study the effects of stack frame layout in larger programs, and engineering work is needed to build these features into existing toolchains.

Of course, we must be cautious to optimize where it matters. While different allocation sizes reduced bit flips relative to each other, the overall effect was minor compared to the savings gained in other data structures. In fact, the reduction in allocation size from 48 to 40 bytes in `xrbt` actually *increased* bit flips in calls to `malloc`, but this increase is dwarfed by the savings from the XOR pointers. Additionally, the hash table saw a relatively small saving compared to other data structures since it already flipped a minimal number of bits in the average case; red-black trees often do more work during *each* update operation, resulting in a number of pointer updates. Hash tables often do their "rebalancing" during a single rehash operation; perhaps bit flip optimization for hash tables should focus on these operations, something we plan to investigate in the future.

CACHE EFFECTS The data structures we tested all had the same behavior—a warm-up period where the cache system absorbed many of the writes followed by a period of proportional increasing of bit flips as the number of update operations increased. We must keep this in-mind when evaluating data structures for bit flips, since we must ensure that the ranges of inputs we test reach the expected scale for our data structures, or we may be blind to its true behavior. The cache size affects this, of course, but it does so in a predictable way in the case of `xrbt`, with only the warm-up period being extended by an amount proportional to cache size. Of course, the behavior may be heavily dependent on write patterns. Thus, we recommend further experiments and that system designers take caches into account when evaluating bit flip behavior of their systems.

The cache additionally affects the read amplification seen in XOR linked lists, wherein the XOR linked list implementation issues more reads than a doubly-linked list implementation. However, the reads that make it to memory are the same between the two, indicating that those extra reads are always in-cache. The resultant write reduction and bit flip reduction is well worth the cost since a read from cache is significantly cheaper than a write to memory.

A.7 FUTURE WORK

Although we covered a range of different data structures, there are many more used in storage systems that should be examined, such as B-trees [7] and LSM-trees [92], both to understand their bit flipping behavior as compared to other data structures and to examine for potential optimizations. In addition to data structures, different algorithms such as sorting can be evaluated for bit flips. Though this may come down to data movement minimization, there may be optimizations in locality that could affect bit flips.

While data structure and algorithm evaluation can provide system designers with insights for how to reduce bit flips, examining bit flips in a large system, including one that properly implements consistency and our suggested stack frame modifications (perhaps through compiler modification), would be worthwhile. There are a number of NVM-based key-value stores [125]; comparing them for bit flips could demonstrate the benefits of some designs over others.

Studying bit flips directly is a good metric for understanding power consumption and wear, but a better understanding through the evaluation of real NVM would be illuminating. The power study discussed earlier was derived from a number of research papers that give approximate numbers or estimates. On a real system, we could *measure* power consumption, and cooperation with vendors may enable accurate studies of wear caused by bit flips. Additionally, some technologies (*e.g.*, PCM) have a disparity between writing a 1 or a 0, something that could be leveraged by software (in cooperation with hardware) to further optimize power use.

A.8 CONCLUSION

The pressures from new storage hardware trends compel us to explore new optimization goals as NVM becomes more common as a persistent store; the read/write asymmetry of NVM must be addressed by reducing bit flips. As we showed, the number of raw writes is not always a faithful proxy for the number of bit flips, so simple techniques that minimize writes overall may be ineffective. At the OS level, we can reconsider memory allocator design to minimize bit flips as pointers are written. Different data structures use and write pointers in different ways, leading to different tradeoffs for data structures when considering NVM applications. At the compiler level, we show that careful layout of stack frames can have a significant impact on bit flips during program operation. Since it can be challenging to reason directly about how application-level writes translate to raw writes due to the compiler and caches, more sophisticated profiling tools are needed to help navigate the tradeoffs between performance, consistency, power use, and wear-out.

Most importantly, we demonstrated the value of reasoning at the *application level* about bit flips, reducing bit flips by $1.13 - 3.56\times$ with minor code changes, no significant increase in complexity, and little performance loss. The data structures we studied had novel implementations, but were *algorithmically* the same as their standard implementations; yet we still saw dramatic improvements with little effort. This indicates that reasoning about bit flips in software can yield significant improvements over in-hardware solutions and opens the door for additional research at a variety of levels of the stack for bit flip reduction. These techniques translate directly to power saving and lifetime improvements, both important optimizations for early adoption of new storage trends that will have lasting impact on systems, applications, and hardware.

BIBLIOGRAPHY

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. “Mach: A New Kernel Foundation for UNIX Development.” In: *Proceedings of the Summer 1986 USENIX Technical Conference*. USENIX. Atlanta, GA, 1986, pp. 93–112. URL: http://www.ssrc.ucsc.edu/PaperArchive/accetta-usenix86s.pdf.

[2] Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. “LiFS: An Attribute-Rich File System for Storage Class Memories.” In: *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*. IEEE. College Park, MD, May 2006. URL: http://www.ssrc.ucsc.edu/Papers/ames-mss06.pdf.

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. North Charleston, SC, USA: CreateSpace Independent Publishing Platform, 2018. ISBN: 198508659X.

[4] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. “Operating System Implications of Fast, Cheap, Non-Volatile Memory.” In: *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS ’11)*. May 2011. URL: http://www.ssrc.ucsc.edu/PaperArchive/bailey-hotos11.pdf.

[5] Piotr Balcer. *An introduction to pmemobj (part 1) - accessing the persistent memory*. https://pmem.io/2015/06/13/accessing-pmem.html. 2015.

[6] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. “The Multikernel: A New OS Architecture for Scalable Multicore Systems.” In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP ’09)*. ACM. Big Sky, MT, Oct. 2009, pp. 29–43. URL: http://www.ssrc.ucsc.edu/PaperArchive/baumann-sosp09.pdf.

[7] Rudolf Bayer and Edward McCreight. “Organization and Maintenance of Large Ordered Indices.” In: *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*. SIGFIDET ’70. Houston, Texas: ACM, 1970,

pp. 107–141. DOI: 10.1145/1734663.1734671. URL: http://doi.acm.org/10.1145/1734663.1734671.

[8] F. Bedeschi, C. Resta, O. Khouri, E. Buda, L. Costa, M. Ferraro, F. Pellizzer, F. Ottogalli, A. Pirovano, and M. Tosi. "An 8MB demonstrator for high-density 1.8 V phase-change memories." In: *Symposium on VLSI Circuits 2004 Digest of Technical Papers*. IEEE, 2004, pp. 442–445.

[9] A. Bensoussan, C. T. Clingen, and R. C. Daley. "The Multics Virtual Memory: Concepts and Design." In: *Proceedings of the 2nd ACM Symposium on Operating Systems Principles (SOSP '69)*. 1969.

[10] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. "Extensibility, Safety, and Performance in the SPIN Operating System." In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. Dec. 1995. URL: http://www.ssrc.ucsc.edu/PaperArchive/bershad-sosp95.pdf.

[11] Nathan Binkert et al. "The Gem5 Simulator." In: *SIGARCH Computer Architecture News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. URL: http://doi.acm.org/10.1145/2024716.2024718.

[12] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. "Wedge: Splitting Applications into Reduced-privilege Compartments." In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*. San Francisco, California: USENIX Association, 2008, pp. 309–322. URL: http://dl.acm.org/citation.cfm?id=1387589.1387611.

[13] Daniel Bittman, Matthew Gray, Justin Raizes, Sinjoni Mukhopadhyay, Matt Bryson, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. "Designing Data Structures to Minimize Bit Flips on NVM." In: *Proceedings of the 7th IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA 2018)*. Aug. 2018. URL: http://www.ssrc.ucsc.edu/bittman-nvmsa18.pdf.

[14] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. "Web caching and Zipf-like distributions: evidence and implications." In: *In Proceedings of Conference on Computer Communications, IEEE INFOCOM '99*. Vol. 1. Mar. 1999, pp. 126–134. DOI: 10.1109/INFCOM.1999.749260.

[15] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. "Overview of candidate device technologies for storage-class memory." In: *IBM Journal of Research and Development* 52.4/5 (July 2008), pp. 449–464. URL: http://www.ssrc.ucsc.edu/PaperArchive/burr-ibmjrd08.pdf.

[16] Adrian M. Caulfield, Arup De, Joel Coburn, Todor Mollov, Rajesh Gupta, and Steven Swanson. "Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories." In: *Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*. 2010, pp. 385–395. URL: http://www.ssrc.ucsc.edu/PaperArchive/caulfield-micro10.pdf.

[17] Robert Cervero. "Road Expansion, Urban Growth, and Induced Travel: A Path Analysis." In: *Journal of the American Planning Association* 69.2 (2003), pp. 145–163. DOI: 10.1080/01944360308976303. eprint: https://doi.org/10.1080/01944360308976303. URL: https://doi.org/10.1080/01944360308976303.

[18] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. "Managing Energy and Server Resources in Hosting Centres." In: *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*. Banff, Canada, Oct. 2001, pp. 103–116. URL: http://www.ssrc.ucsc.edu/PaperArchive/chase-sosp01.pdf.

[19] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. "Sharing and protection in a single-address-space operating system." In: *ACM Transactions on Computer Systems* 12.4 (Nov. 1994), pp. 271–307. URL: http://www.ssrc.ucsc.edu/PaperArchive/chase-tocs94.pdf.

[20] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. "Efficient Support of Position Independence on Non-volatile Memory." In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*. Cambridge, Massachusetts: ACM, 2017, pp. 191–203. ISBN: 978-1-4503-4952-9. URL: http://doi.acm.org/10.1145/3123939.3124543.

[21] Shimin Chen, Phillip B. Gibbons, and Suman Nath. "Rethinking Database Algorithms for Phase Change Memory." In: *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research*. Jan. 2011, pp. 21–31. URL: https://www.microsoft.com/en-us/research/publication/rethinking-database-algorithms-for-phase-change-memory/.

[22] Sangyeun Cho and Hyunjin Lee. “Flip-N-Write: a simple deterministic technique to improve PRAM write performance, energy and endurance.” In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 2009, pp. 347–357.

[23] Howard Chu and Symas. *Lightning Memory-Mapped Database (part of the OpenLDAP project)*. `https://symas.com/lmdb/`.

[24] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. “NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories.” In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’11)*. Mar. 2011, pp. 105–118. URL: `http://www.ssrc.ucsc.edu/PaperArchive/coburn-asplos11.pdf`.

[25] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. “Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components.” In: *Proceedings of SIGMOD 2015 (Industrial Track)*. June 2015. URL: `http://www.ssrc.ucsc.edu/PaperArchive/colgrove-sigmod15.pdf`.

[26] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. “Benchmarking Cloud Serving Systems with YCSB.” In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC ’10)*. Indianapolis, Indiana, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: `10.1145/1807128.1807152`. URL: `http://doi.acm.org/10.1145/1807128.1807152`.

[27] Fernando J. Corbató and Victor A. Vyssotsky. “Introduction and overview of the Multics system.” In: *Proceedings of the November 30 -- December 1, 1965, fall joint computer conference, part I*. ACM, 1965, pp. 185–196. URL: `http://dl.acm.org/citation.cfm?id=1463912` (visited on 12/10/2016).

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[29] Andrew Crotty, Viktor Leis, and Andrew Pavlo. “Are You Sure You Want to Use MMAP in Your Database Management System?” In: *CIDR 2022, Conference on Innovative Data Systems Research*. 2022.

[30] Abdul Dakkak, Cheng Li, Simon Garcia De Gonzalo, Jinjun Xiong, and Wen-mei Hwu. “Trims: Transparent and isolated model sharing for low latency deep learning inference in function-as-a-service.” In: IEEE CLOUD’19.

[31] Robert C. Daley and Jack B. Dennis. “Virtual Memory, Processes, and Sharing in MULTICS.” In: *Communications of the ACM* 11.5 (May 1968), pp. 306–312. URL: http://www.ssrc.ucsc.edu/PaperArchive/daley-cacm68.pdf.

[32] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. “The Clouds Distributed Operating System.” In: *IEEE Computer* (Nov. 1991). URL: http://www.ssrc.ucsc.edu/PaperArchive/dasgupta-computer91.pdf.

[33] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. “Grasshopper: An Orthogonally Persistent Operating System.” In: *Computer Systems* 7.3 (June 1994), pp. 289–312. ISSN: 0895-6340. URL: http://dl.acm.org/citation.cfm?id=198008.198009.

[34] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. “Grasshopper: An Orthogonally Persistent Operating System.” In: *Computer Systems* 7.3 (June 1994), pp. 289–312. ISSN: 0895-6340. URL: http://dl.acm.org/citation.cfm?id=198008.198009.

[35] Biplob Debnath, Sudipta Sengupta, and Jin Li. “FlashStore: High Throughput Persistent Key-Value Store.” In: *Proceedings of the 36th Conference on Very Large Databases (VLDB ’10)*. Sept. 2010. URL: http://www.ssrc.ucsc.edu/PaperArchive/debnath-vldb10.pdf.

[36] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. “PDRAM: A hybrid PRAM and DRAM main memory system.” In: *Proceedings of the 46th IEEE Design Automation Conference (DAC ’09)*. IEEE, 2009, pp. 664–669.

[37] Xiangyu Dong, Cong Xu, Yuan Xie, and N. P. Jouppi. “NVSim: a Circuit-Level Performance, Energy, and area Model for Emerging Nonvolatile Memory.” In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 31.7 (July 2012). ISSN: 0278-0070, 1937-4151. DOI: 10.1109/TCAD.2012.2185930. URL: http://ieeexplore.ieee.org/document/6218223/ (visited on 03/05/2018).

[38] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. "System Software for Persistent Memory." In: *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*. Apr. 2014. URL: http://www.ssrc.ucsc.edu/PaperArchive/dulloor-eurosys14.pdf.

[39] Jeremy Condit and Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. "Better I/O Through Byte-Addressable, Persistent Memory." In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. Big Sky, MT, Oct. 2009, pp. 133–146. URL: http://www.ssrc.ucsc.edu/PaperArchive/condit-sosp09.pdf.

[40] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. "Maglev: A Fast and Reliable Software Network Load Balancer." In: *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Mar. 2016. ISBN: 978-1-931971-29-4.

[41] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. "SpaceJMP: Programming with Multiple Virtual Address Spaces." In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Atlanta, Georgia, USA: ACM, 2016, pp. 353–368. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872366. URL: http://doi.acm.org/10.1145/2872362.2872366.

[42] Dawson R Engler, Sandeep K Gupta, and M Frans Kaashoek. "AVM: Application-level virtual memory." In: *Fifth Workshop on Hot Topics in Operating Systems (HotOS '95)*. IEEE. 1995, pp. 72–77.

[43] Dawson R Engler and M Frans Kaashoek. "Exterminate all operating system abstractions." In: *Fifth Workshop on Hot Topics in Operating Systems (HotOS '95)*. IEEE. 1995, pp. 78–83.

[44] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. "Exokernel: An Operating System Architecture for Application-Level Resource Management." In: *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*. Dec. 1995, pp. 251–266. URL: http://www.ssrc.ucsc.edu/PaperArchive/engler-sosp95.pdf.

[45] Hewlett Packard Enterprise. *YCSB-C*. https://github.com/HewlettPackard/meadowlark/tree/master/extra/YCSB-C https://github.com/basicthinker/YCSB-C. 2018.

[46] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. "Beyond Processor-centric Operating Systems." In: *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*. Kartause Ittingen, Switzerland: USENIX Association, May 2015. URL: https://www.usenix.org/conference/hotos15/workshop-program/presentation/faraboschi.

[47] GR Fox, F Chu, and T Davenport. "Current and future ferroelectric nonvolatile memory technology." In: *Journal of Vacuum Science & Technology B: Microelectronics and Nanometer Structures Processing, Measurement, and Phenomena* 19.5 (2001), pp. 1967–1971.

[48] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. "Semantic File Systems." In: *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*. ACM. Oct. 1991, pp. 16–25. URL: http://www.ssrc.ucsc.edu/PaperArchive/gifford-sosp91.pdf.

[49] Burra Gopal and Udi Manber. "Integrating Content-Based Access Mechanisms with Hierarchical File Systems." In: *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*. Feb. 1999, pp. 265–278. URL: http://www.ssrc.ucsc.edu/PaperArchive/gopal-osdi99.pdf.

[50] Kevin M. Greenan and Ethan L. Miller. "PRIMS: Making NVRAM Suitable for Extremely Reliable Storage." In: *Proceedings of the Third Workshop on Hot Topics in System Dependability (HotDep '07)*. June 2007. URL: http://www.ssrc.ucsc.edu/Papers/greenan-hotdep07.pdf.

[51] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. "Software Persistent Memory." In: *Proceedings of the 2012 USENIX Annual Technical Conference*. 2012. URL: http://www.ssrc.ucsc.edu/PaperArchive/guerra-atc12.pdf.

[52] Miseon Han, Youngsun Han, Seon Wook Kim, Hokyoon Lee, and Il Park. "Content-Aware Bit Shuffling for Maximizing PCM Endurance." In: *ACM Transactions on Design Automation of Electronic Systems* 22.3 (May 2017), 48:1–48:26. ISSN: 1084-4309. DOI: 10.1145/3017445. URL: http://doi.acm.org/10.1145/3017445.

[53] Gernot Heiser and Kevin Elphinstone. “L4 Microkernels: The Lessons from 20 Years of Research and Deployment.” In: *ACM Trans. Comput. Syst.* 34.1 (Apr. 2016). ISSN: 0734-2071. DOI: 10.1145/2893177. URL: https://doi.org/10.1145/2893177.

[54] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. *Mungi: A Distributed Single Address-Space Operating System*. Tech. rep. 9314. School of Computer Science and Engineering, University of New South Wales, Nov. 1993. URL: http://www.ssrc.ucsc.edu/PaperArchive/heiser-scse9314.pdf.

[55] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. “Mesos: A Platform for Fine-grained Resource Sharing in the Data Center.” In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI ’11)*. Boston, MA: USENIX, 2011, pp. 295–308. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488.

[56] Jeffrey Hollingsworth and Ethan Miller. “Using Content-Derived Names for Configuration Management.” In: *Proceedings of the 1997 Symposium on Software Reusability (SSR ’97)*. IEEE. Boston, MA, May 1997, pp. 104–109. URL: http://www.ssrc.ucsc.edu/~elm/Papers/ssr97.pdf.

[57] Antony L. Hosking and J. Eliot B. Moss. “Object Fault Handling for Persistent Programming Languages: A Performance Evaluation.” In: *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA ’93)*. Washington, D.C., USA: ACM, 1993, pp. 288–303. ISBN: 0-89791-587-9. DOI: 10.1145/165854.165907. URL: http://doi.acm.org/10.1145/165854.165907.

[58] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibrod. “Log-Structured Non-Volatile Main Memory.” In: *Proceedings of the 2017 USENIX Annual Technical Conference*. Santa Clara, CA, June 2017, pp. 703–717. URL: http://www.ssrc.ucsc.edu/PaperArchive/hu-atc17.pdf.

[59] *IBM MQ*. https://www-03.ibm.com/software/products/en/ibm-mq. 2019.

[60] Intel Newsroom. *Intel and Micron Produce Breakthrough Memory Technology*. http://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/; Accessed 2019-01-10. 2015.

[61] Joseph Izraelevitz et al. "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module." In: *arXiv* abs/1903.05714 (2019). arXiv: 1903.05714. URL: http://arxiv.org/abs/1903.05714.

[62] Adam N Jacobvitz, Robert Calderbank, and Daniel J Sorin. "Coset coding to extend the lifetime of memory." In: *Proceedings of High Performance Computer Architecture (HPCA '13)*. IEEE. 2013, pp. 222–233.

[63] Hrishikesh Jayakumar, Kangwoo Lee, Woo Suk Lee, Arnab Raha, Younghyun Kim, and Vijay Raghunathan. "Powering the Internet of Things." In: *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED '14)*. La Jolla, California, USA: ACM, 2014, pp. 375–380. ISBN: 978-1-4503-2975-0. DOI: 10.1145/2627369.2631644. URL: http://doi.acm.org/10.1145/2627369.2631644.

[64] Hrishikesh Jayakumar, Arnab Raha, and Vijay Raghunathan. "QuickRecall: A low overhead HW/SW approach for enabling computations across power cycles in transiently powered computers." In: *Proceedings of the 27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*. IEEE. 2014, pp. 330–335.

[65] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. "Fine-grained Mobility in the Emerald System." In: *ACM Transactions on Computer Systems* 6.1 (Feb. 1988), pp. 109–133. ISSN: 0734-2071. DOI: 10.1145/35037.42182. URL: http://doi.acm.org/10.1145/35037.42182.

[66] Eric Jul and Bjarne Steensgaard. "Implementation of distributed objects in Emerald." In: *Proceedings of International Workshop on Object Orientation in Operating Systems*. IEEE, 1991, pp. 130–132.

[67] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. "Application Performance and Flexibility on Exokernel Systems." In: *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*. Saint Malo, France: ACM, 1997, pp. 52–65. ISBN: 0-89791-916-5. DOI: 10.1145/268998.266644. URL: http://doi.acm.org/10.1145/268998.266644.

[68] Peter Kairouz et al. "Advances and Open Problems in Federated Learning." In: *CoRR* abs/1912.04977 (2019).

[69] T. Kawahara. "Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing." In: *IEEE Design and Test of Computers* 28.1 (Jan. 2011), pp. 52–63. ISSN: 0740-7475. DOI: 10.1109/MDT.2010.97.

[70] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution." In: *arXiv preprint arXiv:1801.01203* (2018).

[71] Eddie Kohler. *Left-Leaning Red-Black Trees Considered Harmful.* http://read.seas.harvard.edu/~kohler/notes/llrb.html. Accessed 2018-09-22.

[72] Jay Kreps, Neha Narkhede, and Jun Rao. "Kafka: A distributed messaging system for log processing." In: *Proceedings of The 6th International Workshop on Networking Meets Databases (NetDB'11).* June 2011. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf.

[73] Orran Krieger et al. "K42: Building a Complete Operating System." In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06).* Leuven, Belgium: ACM, 2006, pp. 133–145. ISBN: 1-59593-322-0. DOI: 10.1145/1217935.1217949. URL: http://doi.acm.org/10.1145/1217935.1217949.

[74] Hugh C. Lauer and Roger M. Needham. "On the Duality of Operating System Structures." In: *ACM SIGOPS Operating Systems Review* 13.2 (Apr. 1979), pp. 3–19. ISSN: 0163-5980. DOI: 10.1145/850657.850658. URL: http://doi.acm.org/10.1145/850657.850658.

[75] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. "Architecting phase change memory as a scalable dram alternative." In: *ACM SIGARCH Computer Architecture News.* Vol. 37. ACM, 2009, pp. 2–13.

[76] Dokeun Lee and Youjip Won. "Bootless Boot: Reducing Device Boot Latency with Byte Addressable NVRAM." In: *2013 International Conference on High Performance Computing.* Nov. 2013. URL: http://www.ssrc.ucsc.edu/PaperArchive/lee-hpcc13.pdf.

[77] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. "Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs." In: *2019 USENIX Annual Technical Conference (USENIX ATC 19).* Renton, WA: USENIX Association, July 2019, pp. 603–616. ISBN: 978-1-939133-

03-8. URL: https://www.usenix.org/conference/atc19/presentation/lee-gyusun.

[78] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown." In: *arXiv preprint arXiv:1801.01207* (2018).

[79] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. "Light-Weight Contexts: An OS Abstraction for Safety and Performance." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. GA: USENIX Association, 2016, pp. 49–64. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/litton.

[80] Youyou Lu, Jiwu Shu, and Long Sun. "Blurred Persistence: Efficient Transactions in Persistent Memory." In: *ACM Transactions on Storage* 12.1 (Jan. 2016). URL: http://www.ssrc.ucsc.edu/PaperArchive/lu-tos16.pdf.

[81] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. "Loose-ordering consistency for persistent memory." In: *Proceedings of the 32nd IEEE International Conference on Computer Design (ICCD '14)*. IEEE. 2014, pp. 216–223.

[82] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. "Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory." In: *Proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '17)*. Santa Clara, CA: USENIX Association, 2017. URL: https://www.usenix.org/conference/hotstorage17/program/presentation/marathe.

[83] Alex Markuze, Adam Morrison, and Dan Tsafrir. "True IOMMU Protection from DMA Attacks: When Copy is Faster Than Zero Copy." In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. Atlanta, Georgia, USA: ACM, 2016, pp. 249–262. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872379. URL: http://doi.acm.org/10.1145/2872362.2872379.

[84] Pankaj Mehra and Samuel Fineberg. "Fast and Flexible Persistence: The Magic Potion for Fault-Tolerance, Scalability and Performance in Online Data Stores." In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*. Jan. 2004. DOI: 10.1109/IPDPS.2004.1303232.

[85] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. "A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory." In: *5th Workshop on Energy-Efficient Design (WEED '13)*. June 2013. URL: http://www.ssrc.ucsc.edu/PaperArchive/meza-weed13.pdf.

[86] Donald Miller and Alan Skousen. "The Sombrero Distributed Single Address Space Operating System." In: *ACM SIGOPS Operating Systems Review* 34.2 (Apr. 2000), p. 37. ISSN: 0163-5980. DOI: 10.1145/346152.346257. URL: http://doi.acm.org/10.1145/346152.346257.

[87] Mark S. Miller, Ka-Ping Yee, and Jonathan Shapiro. "Capability myths demolished." In: (2003). Technical report; Johns Hopkins University Systems Research Laboratory. URL: %5Curl%7Bhttp://www.erights.org/elib/capability/duals%7D.

[88] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge; NY: Cambridge University Press, 1995.

[89] Dushyanth Narayanan and Orion Hodson. "Whole-System Persistence." In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*. Mar. 2012, pp. 401–500. URL: http://www.ssrc.ucsc.edu/PaperArchive/narayanan-asplos12.pdf.

[90] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. "Reducing NVM Writes with Optimized Shadow Paging." In: *Proceedings of the 10th Workshop on Hot Topics in Storage and File Systems (HotStorage '18)*. July 2018. URL: http://www.ssrc.ucsc.edu/ni-hotstorage18.pdf.

[91] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. "SSP: Eliminating Redundant Writes in Failure-Atomic NVRAMs via Shadow Sub-Paging." In: *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*. Oct. 2019.

[92] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The Log-structured Merge-tree (LSM-tree)." In: *Acta Informatica* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903. DOI: 10.1007/s002360050048. URL: http://dx.doi.org/10.1007/s002360050048.

[93] Matheus Ogleari, Ethan L. Miller, and Jishen Zhao. "Steal but no force: Efficient Hardware-driven Undo+Redo Logging for Persistent Memory Systems." In: *Proceedings of the 24th International Symposium on High-Performance Computer Architecture (HPCA*

2018). Feb. 2018. URL: http://www.ssrc.ucsc.edu/ogleari-hpca18.pdf.

[94] John Ousterhout et al. "The RAMCloud Storage System." In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015), 7:1–7:55. ISSN: 0734-2071. URL: http://doi.acm.org/10.1145/2806887.

[95] Yoann Padioleau and Olivier Ridoux. "A Logic File System." In: *Proceedings of the 2003 USENIX Annual Technical Conference*. San Antonio, TX, June 2003, pp. 99–112. URL: http://www.ssrc.ucsc.edu/PaperArchive/padioleau-usenix03.pdf.

[96] Aleatha Parker-Wood, Darrell D. E. Long, Ethan L. Miller, Philippe Rigaux, and Andy Isaacson. "A File By Any Other Name: Managing File Names with Metadata." In: *Proceedings of the 7th Annual International Systems and Storage Conference (SYSTOR '14)*. June 2014. URL: http://www.ssrc.ucsc.edu/Papers/parkerwood-systor14.pdf.

[97] *PMDK*. https://pmem.io/pmdk/.

[98] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. "Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling." In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2009.

[99] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. "Scalable high performance main memory system using phase-change memory technology." In: *Proceedings of the 36th annual international symposium on Computer architecture (ICSA '09)*. 2009, pp. 24–33.

[100] S. Raoux et al. "Phase-change random access memory: A scalable technology." In: *IBM Journal of Research and Development* 52.4/5 (July 2008), pp. 465–480. URL: http://www.ssrc.ucsc.edu/PaperArchive/raoux-ibmjrd08.pdf.

[101] Timothy Roscoe. "Linkage in the Nemesis single address space operating system." In: *ACM SIGOPS Operating Systems Review* 28.4 (Oct. 1994), pp. 48–55. URL: http://www.ssrc.ucsc.edu/PaperArchive/roscoe-osr94.pdf.

[102] Andy Rudoff et al. *Persistent Memory Programming Library*. http://pmem.io/nvml/. 2017.

[103] Robert Sedgewick and Leonidas. J. Guibas. "A dichromatic framework for balanced trees." In: *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (SFCS '78)*. Vol. 00. Oct. 1978, pp. 8–21. DOI: 10.1109/SFCS.1978.3. URL: doi.ieeecomputersociety.org/10.1109/SFCS.1978.3.

[104] Seyed Mohammad Seyedzadeh, Rakan Maddah, Donald Kline, Alex K Jones, and Rami Melhem. "Improving bit flip reduction for biased and random data." In: *IEEE Transactions on Computers* 65.11 (2016), pp. 3345–3356.

[105] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. "Distributed Shared Persistent Memory." In: *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. Santa Clara, California: Association for Computing Machinery, 2017, pp. 323–337. ISBN: 9781450350280. DOI: 10.1145/3127479.3128610. URL: https://doi.org/10.1145/3127479.3128610.

[106] Jonathan S. Shapiro and Jonathan Adams. "Design Evolution of the EROS Single-Level Store." In: *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX. Monterey, CA, June 2002, pp. 59–72. URL: http://www.ssrc.ucsc.edu/PaperArchive/shapiro-usenix02.pdf.

[107] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: A Fast Capability System." In: *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. Charleston, South Carolina, USA: ACM, 1999, pp. 170–185. ISBN: 1-58113-140-2. DOI: 10.1145/319151.319163. URL: http://doi.acm.org/10.1145/319151.319163.

[108] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. "A comprehensive study of convergent and commutative replicated data types." In: (2011).

[109] Eugene Shekita and Michael Zwilling. *Cricket: A Mapped, Persistent Object Store*. Tech. rep. 956. University of Wisconsin, Aug. 1990. URL: http://www.ssrc.ucsc.edu/PaperArchive/shekita-uw-tr956.pdf.

[110] Shyh-Shyuan Sheu et al. "Fast-Write Resistive RAM (RRAM) for Embedded Applications." In: *IEEE Design & Test of Computers* (Jan. 2011), pp. 64–71. URL: http://www.ssrc.ucsc.edu/PaperArchive/shen-dtc11.pdf.

[111] Nikita Shirokov and Ranjeeth Dasineni. *Open-sourcing Katran, a scalable network load balancer — Facebook Engineering*. https://code.fb.com/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. May 2018.

[112] Mohammed Shoeybi. *Training Multi-billion parameter models in Megatron*. https://hotchips.org/archives/hc32/. 2020.

[113] Abhishek Singh, Praneeth Vepakomma, Otkrist Gupta, and Ramesh Raskar. "Detailed comparison of communication efficiency of split learning and federated learning." In: *arXiv preprint arXiv:1909.09145* (2019).

[114] Prokash Sinha. “A Memory-Efficient Doubly Linked List.” In: *Linux Journal* 129 (2004). http://www.linuxjournal.com/article/6828.

[115] Alan Skousen and Donald Miller. “Using a single address space operating system for distributed computing and high performance.” In: *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC ’99)*. Feb. 1999, pp. 8–14. URL: skousen-ipccc99.pdf.

[116] *SQLite*. https://www.sqlite.org/index.html.

[117] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. “The missing memristor found.” In: *Nature* 453 (May 2008), pp. 80–83. URL: http://www.ssrc.ucsc.edu/PaperArchive/strukov-nature08.pdf.

[118] *The musl C Library*. https://musl.libc.org/.

[119] *The Rust Programming Language*. https://www.rust-lang.org/.

[120] *TIBCO Rendezvous*. https://www.tibco.com/products/tibco-rendezvous. 2019.

[121] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. “The Aurora Single Level Store Operating System.” In: *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. SOSP ’21. Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 788–803. ISBN: 9781450387095. DOI: 10.1145/3477132.3483563. URL: https://doi.org/10.1145/3477132.3483563.

[122] Haris Volos, Andres Jaan Tack, and M. Swift. “Mnemosyne: Lightweight Persistent Memory.” In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’11)*. Mar. 2011. URL: http://www.ssrc.ucsc.edu/PaperArchive/volos-asplos11.pdf.

[123] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. “Hardware Supported Persistent Object Address Translation.” In: *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO ’17)*. Cambridge, Massachusetts: ACM, 2017, pp. 800–812. ISBN: 978-1-4503-4952-9. DOI: 10.1145/3123939.3123981. URL: http://doi.acm.org/10.1145/3123939.3123981.

[124] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, C. Pierson, and Fred Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System." In: *Communications of the ACM* 17.6 (June 1974), pp. 337–345. ISSN: 0001-0782. DOI: 10.1145/355616.364017.

[125] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems." In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 349–362. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/xia.

[126] Jian Xu and Steven Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories." In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST '16)*. Santa Clara, CA: USENIX Association, 2016, pp. 323–338. ISBN: 978-1-931971-28-7. URL: http://dl.acm.org/citation.cfm?id=2930583.2930608.

[127] Jian Xu and Steven Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories." In: *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*. Feb. 2016. URL: http://www.ssrc.ucsc.edu/PaperArchive/xu-fast16.pdf.

[128] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. "Performance Analysis of NVMe SSDs and Their Implication on Real World Databases." In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR '15. Haifa, Israel: Association for Computing Machinery, 2015. ISBN: 9781450336079. DOI: 10.1145/2757667.2757684. URL: https://doi.org/10.1145/2757667.2757684.

[129] B. D. Yang, J. E. Lee, J. S. Kim, J. Cho, S. Y. Lee, and B. G. Yu. "A Low Power Phase-Change Random Access Memory using a Data-Comparison Write Scheme." In: *Proceedings of IEEE International Symposium on Circuits and Systems*. May 2007. DOI: 10.1109/ISCAS.2007.377981.

[130] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. "A durable and energy efficient main memory using phase change memory technology." In: *Proceedings of the 36th International Symposium on Computer Architecture*. 2009, pp. 14–23. URL: http://www.ssrc.ucsc.edu/PaperArchive/zhou-isca09.pdf.