Operator, Number Please: Mediating Access to Shared Resources for Efficiency and Isolation

by

Mihir Sudarshan Nanavati

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

January 2019

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Operator, Number Please: Mediating Access to Shared Resources for Efficiency and Isolation

submitted by **Mihir Sudarshan Nanavati** in partial fulfillment of the requirements for the degree of **Doctor of Philosophy** in **Computer Science**.

Examining Committee:

Andrew Warfield, Computer Science

Co-supervisor

William Aiello, Computer Science

Co-supervisor

Margo Seltzer, Computer Science

University Examiner

Alexandra Fedorova, Electrical and Computer Engineering

University Examiner

Donald E. Porter, Computer Science

External Examiner

Additional Supervisory Committee Members:

Norm Hutchinson, Computer Science

Supervisory Committee Member

Abstract

The performance density of modern hardware has forced the sharing of hardware resources across applications for better utilization and efficiency. Shared infrastructure, however, weakens isolation and risks interference, which can result in degraded performance and security breaches. This thesis explores the tension between isolation and sharing with three prototype systems: Xoar, Plastic, and Decibel. All three of these systems demonstrate the value of software mediation in providing isolation on shared hardware without sacrificing either hardware resource utilization or the performance of the underlying devices.

Xoar, Plastic, and Decibel provide isolation for different hardware resources: Xoar strengthens isolation between virtual machines, thereby allowing underutilized processors to be shared; Plastic transparently mitigates poor cache utilization and the performance artifacts caused by insufficient cache line isolation across cores; and Decibel provides isolation in shared non-volatile storage and guarantees throughput, even in the face of competing workloads.

Lay Summary

Computer hardware has become significantly faster and more capable over the last decade. At the same time, the requirements of most software applications have not increased at nearly the same rate; consequently, a modern high-performance server running only a single application will remain mostly idle. To combat this inefficiency, modern hardware is usually shared across multiple applications. Unfortunately, while this sharing increases efficiency, it also increases the potential for applications to interfere with each other. This thesis explores techniques to provide isolation on top of shared hardware and describes systems that enable high resource utilization on modern servers while benefiting applications by providing them access to shared and isolated hardware resources at performance comparable to dedicated hardware.

Preface

All the systems described in this thesis have been published individually at peer-reviewed academic conferences. The corresponding chapters are the contents of these publications very lightly edited for formatting purposes.

Chapter 2 was originally published as "Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor" at SOSP 2011 [35]. It is joint work with Patrick Colp, Jun Zhu, William Aiello, George Coker, Tim Deegan, Peter Loscocco, and Andrew Warfield. The author did not lead this project, but was heavily involved in both the development of the system and the writing of the paper.

Chapter 3 was originally published as "Whose Cache Line Is It Anyway? Operating System Support for Live Detection and Repair of False Sharing" at EuroSys 2013 [117]. It is joint work with Mark Spear, Nathan Taylor, Shriram Rajagopalan, Dutch T. Meyer, William Aiello, and Andrew Warfield. The author lead the design and implementation of the system and presented the work at the conference.

Chapter 4 was originally published as "Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage" at NSDI 2017 [118]. It is joint work with Jake Wires and Andrew Warfield. The author lead the design and implementation of the system and presented the work at the conference.

Table of Contents

List of Tables

List of Figures

Acknowledgements

There are a number of people to whom I owe a debt of gratitude for making the last several years of graduate school as thoroughly enjoyable as they have been. First off, to my advisors, Andy Warfield and Bill Aiello, who have been excellent at nurturing a student with a short attention span, and have provided me with the right balance of freedom to explore any new topics that catch my fancy and structure to not get overwhelmed with the effort. Andy and Bill have been generous with their time and expertise, have indulged my idiosyncracies with good humour, and have routinely exhibited a greater determination to see me succeed than I have. I am fortunate to have had them on my side and even more fortunate to now consider them friends—I look forward to opportunities to work together in the future.

As the third member of my committee, Norm Hutchinson has been warm and approachable at all times, and has been quick to spot flaws in research ideas and then point them out exceedingly gently and patiently. I would also like to thank my examiners, Sasha Fedorova, Don Porter, and Margo Seltzer, for questions and comments that have improved this thesis.

Over the course of graduate school, I've worked with Tim Deegan (at Citrix), and with Rama Kotla, Andrew Baumann, and Sid Sen (at various times, all at Microsoft Research). Each of these experiences was both hugely educational and also lots of fun, and I am thankful for having had the opportunity to do so. I also spent a couple of delightful years at Coho Data working primarily with Tim Deegan, Geoffrey Lefebvre, Steven Smith, Andy Warfield, and Jake Wires. My thanks to them, and to the rest of the engineering team at Coho Data, for a bunch of great conversations and just

being an all-round fantastic set of folks to work with.

The Systems lab at UBC has provided me with a set of great colleagues and collaborators over the years in Cody Brown, Patrick Colp, Brendan Cully, Sara Dadizadeh, Chris Head, Geoffrey Lefebvre, Jean-Sébastien Légaré, Dutch Meyer, Ryan O'Connor, Shriram Rajagopalan, Mark Spear, alexandru totolici, Nathan Taylor, and Jake Wires. None of the systems described here would exist without their willingness to put up with my constantly bouncing ridiculous schemes off them—I am grateful for the cheer and good company, as well as for all the code that we've written together.

On a personal level, a bunch of friends, whom I hesitate to name individually for fear of embarrassing them, have provided more companionship, encouragement, and moral support, than I'd have thought possible (especially true for those over an instant messaging window). Spread across the world, they've graciously hosted me over the years and have always been up for anything from last minute travel escapades, to trying out outrageous coffee shops (and bakeries), restaurants, and bars, to skiing down mountains and scrambling over rocks, to debating the geopolitics of the world around us (or just debating the merits of a particular sports team instead). You know who you are and my life is richer for your presence.

Finally, to my family who, in addition to being instrumental in getting me into this mess of graduate school in the first place, have been unwavering in their love and support and have gladly provided the necessary combination of nagging and cheerleading in any endeavour of mine. This one truly is for you and I look forward to the next time you ask "Are you done yet?"

To my family, with love.

You all thought this was a good idea.
Now you know better.

Chapter 1

Introduction

Modern server systems are extremely resource-dense. A high-end server today contains tens to hundreds of cores, terabytes of memory, gigabits of network bandwidth, and hundreds of terabytes of storage. As these resources often outstrip the requirements of a single application, the desire for efficient resource utilization necessitates sharing.

The need for sharing is most readily apparent in the case of datacenters. Over the last several years, the demands of providing millions of users with fast, reliable access to large Internet applications, such as Netflix, Dropbox, and Spotify, has lead to these applications being developed as two distinct parts: an application-specific component for handling user requests, and a generic, infrastructure component that manages hardware resources to ensure the performance and availability of the application. This separation has been formalized into separate businesses, with cloud providers, such as Amazon, Microsoft, and Google, embracing the common operational aspects and developing and hosting managed infrastructure which presents the abstraction of infinite, low-latency, always-available resources for these applications to build on.

Public cloud providers host tens of thousands of competing applications which have highly elastic resource requirements, and expect steady performance and availability, even while scaling up and down. Datacenter economics require that resources be shared across these applications, and

that providers rely on statistical multiplexing for efficient resource utilization and to absorb sudden surges in demand for any application. Conversely, shared infrastructure risks interference that results in degraded performance or even a complete denial of service: applications that consume more than their fair share of *any* resource affect *all* other applications sharing those resources. Even more consequentially, information disclosure achieved by exploiting the underlying platform or data leakage via shared hardware channels can impact the confidentiality of applications.

This is not solely an issue at the scale of datacenters. As individual servers become more powerful, sharing resources is inevitable even in small, single-owner deployments. While having a single ownership domain may ameliorate the security concerns around sharing, performance isolation is still necessary to ensure that applications do not inadvertently impact one another due to greedy, wasteful, or simply a flawed use of shared resources.

This represents a Catch-22: on the one hand, it is imperative to share resources as a means of deriving value from fast and expensive hardware resources. On the other, sacrificing isolation between applications through sharing increases the likelihood of outages and breaches. This thesis targets such resource-dense deployments and asserts that *it is the responsibility of the platform to resolve this tension and that it can provide both security and performance isolation on shared hardware, while preserving the performance of the underlying hardware and supporting existing applications with little to no modification, through the proactive management of all shared resources.*

The thesis explores this idea using three projects. Xoar [35] increases security and cross-tenant isolation for co-located virtual machines by reducing the amount of trusted code in the virtualization platform, and by making all resource sharing explicit. Plastic [117] targets compute cycles wasted due to unnecessary memory contention and transparently mitigates false sharing by forcing a cache-aware layout for hot data. Decibel [118] provides isolation in consolidated storage and allows remote tenants to safely share fast, non-volatile devices across remote tenants with low overhead.

The following three chapters are taken from the published versions of the projects described above. As each chapter has been taken from an inde-

pendent conference paper, it describes a system in isolation. The remainder of this chapter is an attempt to contextualize these projects with an eye to the broader research goals of the thesis.

1.1 Xoar *(SOSP 2011)*

Virtualization has been at the forefront of the shift towards consolidated hardware by allowing providers to lease small slices of large-scale datacenter resources to individual customers and to co-locate applications encapsulated within individual virtual machines. Virtualization not only balances the operational efficiency required by providers with the need to present applications the illusion of dedicated hardware, but also allows providers to offer additional functionality, such as the ability to clone existing machines, or automatically migrate them in response to hardware failures. This clean separation of hosted applications from the underlying hardware is responsible for the popularity of virtual machines in cloud computing today.

Underlying this popularity is the trust that the provider's virtualization platform is secure enough to isolate co-located applications—a belief mostly predicated on the relatively modest footprint of processor hypervisors and the narrow interface they expose to virtual machines. Despite this, virtualization platforms remain vulnerable as they are required to trust several components in addition to the hypervisor. As an example, Xen relies on a privileged commodity OS (typically an entire Linux or BSD instance) for hardware support, device emulation, and the administrative and management tool stack. This privileged OS has been a significant source of security disclosures and vulnerabilities.

Xoar is a virtualization platform, based on Xen, that retrofits the modularity and isolation principles used in microkernels onto an existing platform with little performance overhead. Designed as a drop-in replacement for Xen, it does not require existing functionality to be sacrificed, operating systems and applications to be modified, or components of the platform to be rewritten from scratch.

Xoar deconstructs the privileged commodity OS in Xen into a set of

single-purpose components, each of which are restricted to the least privilege necessary, and have well-defined interfaces with tenant virtual machines, making any sharing of components explicit. Xoar allows providers to log and audit tenants relying on these shared services, while tenants are able to precisely specify the degree of sharing they are comfortable with and opt-out of any disagreeable sharing configuration.

1.2 Plastic *(EuroSys 2013)*

Application performance is largely determined by memory latency, with processors remaining idle for increasingly large fractions of execution waiting for code and data to be loaded from memory [48, 82]. This has lead to the development of complex, multi-level memory hierarchies with several different types of caches that allow processors to mask access latencies for frequently accessed data. While the number, size, latency, and exclusivity of caches vary across microarchitectures, they are largely a transparent, hardware-managed performance optimization on modern server processors.

As hardware parallelism continues to increase, the degree of cache contention rises dramatically. Cross-core contention takes multiple forms depending on the workload and the microarchitecture; for instance, different applications may compete for capacity in a shared cache, while multiple applications, or even threads within the same one, may compete for ownership of specific cache lines containing shared data structures. As memory layouts and access patterns affect contention, the hardware is unable to transparently provide either efficient cache utilization or workload isolation, and it becomes the responsibility of the platform to treat caches like other shared hardware resources, and manage them appropriately.

Plastic focuses on a particular class of memory contention called *false sharing*, that occurs due to contention over a single cache line at the microarchitectural level. Architectural instruction sets isolate memory at the granularity of a single byte or word, whereas the cache controller operates on entire cache lines. This granularity mismatch sometimes results in specific data layouts where applications accessing discrete, unshared mem-

ory locations trigger accesses to the same underlying shared cache line and cause microarchitectural contention that impacts performance.

Broadly speaking, there are two approaches to mitigating false sharing: modifying data layouts to remove contention, or modifying access patterns to reduce the frequency of contention to the point that it is no longer detrimental to application performance. Plastic takes the former approach and is capable of rapidly detecting, precisely identifying, and transparently repairing false sharing in unmodified, running applications with little overhead by remapping contended memory regions to independent cache lines.

Fine-grained memory virtualization using hardware extensions has been explored extensively in Mondriaan [162]. Plastic provides a similar generic memory virtualization facility, for sub-page granularity remapping, entirely in software, and fronts it with a memory contention detector. The detection and remapping-based strategy for false sharing adopted by Plastic is not restricted to the platform, and has subsequently been used within language runtimes to resolve instances of false sharing by leveraging the ability of the runtime environment to pad and move existing live data structures [45].

1.3 Decibel *(NSDI 2017)*

Despite the arrival of high-speed, non-volatile storage devices in datacenters, applications today are forced to pick between one of two extremes: managed block storage that is a shared, globally accessible resource, but is extremely limited in the performance guarantees it offers, and direct-attached local storage devices that mirror the performance of the underlying device, but are tightly coupled to a single host, cannot be shared, and are only suitable for temporary storage. While applications have traditionally opted for the simplicity of managed storage, the differences in absolute performance and predictability of these storage classes have resulted in several datacenter infrastructure applications, such as Cassandra and Kafka, opting for direct-attached storage [37, 93, 109], despite its obvious shortcomings.

Disaggregating storage into a rack- or cluster-shared resource implicitly decouples storage from a single host and presents system designers an op-

portunity to offer these applications a useful new class of fast, shared storage. Instead, early designs integrating disaggregated storage have focused entirely on speed and produced specialized systems capable of driving storage devices to the very edge of their performance envelope, at the cost of functionality important for application-facing storage abstractions.

One such casualty is isolation: several of these systems bypass the processor and software stack on the storage server entirely and expose devices directly over low-latency hardware interconnects to applications [5, 30, 46]. By directly sharing physical devices, they lack the ability to interpose on accesses and require additional hardware support and complicated distributed data paths to provide virtualization and performance isolation [68].

Decibel studies the functionality storage abstractions should provide these applications and the feasibility of doing so using commodity networking and storage hardware. It is predicated on the belief that, for multi-tenant storage, as long as performance can be maintained, the operational simplicity of centralizing control and enforcing isolation policies by mediating access to storage using locally-attached compute is valuable.

To achieve this, Decibel proposes a lightweight storage abstraction that resembles a virtualized network-attached disk, but focuses solely on providing isolation for shared devices, and demonstrates a runtime capable of serving this isolated storage to remote tenants within a single rack over commodity Ethernet-based networking at a throughput and latency comparable to local devices.

Chapter 2

Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor

2.1 Introduction

Datacenter computing has shifted the criteria for evaluating system design from one that prioritizes peak capacity and offered load, to one that emphasizes the efficiency with which computing is delivered [4, 9, 155, 160]. This is particularly true for cloud hosting providers, who are motivated to reduce costs and therefore to multiplex and over-subscribe their resources as much as possible while still meeting customer service level objectives (SLOs).

While the efficiency of virtualization platforms remains a primary factor in their commercial success, their administrative features and benefits have been equally important. For example, hardware failures are a fact of life for large hosting environments; such environments rely on functionality such as live VM migration [33] for planned hardware replacements as well as unexpected failures [22, 38]. Hardware diversity is also inevitable in a large hosting facility; the use of hardware emulation and unified virtual devices means that a single VM image can be hosted on hardware throughout the

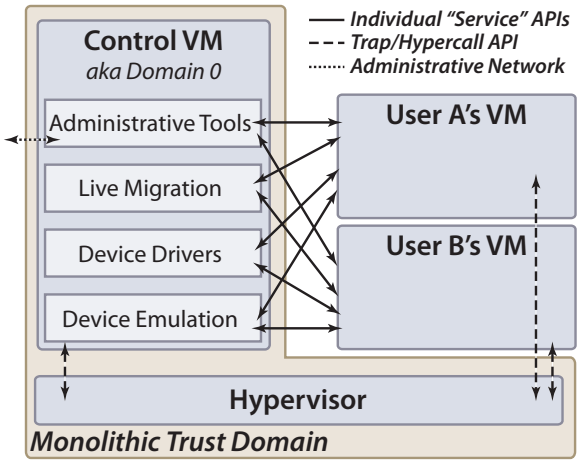

Figure 2.1: The control VM is often a full operating system install, has privilege similar to the hypervisor, and offers multiple services over numerous interfaces to guest VMs.

facility without the need for device driver upgrades within customer VMs. Administrative benefits aside, the largest reason for the success of virtualization may be that it requires little or no change to existing applications. These three factors (resource utilization, administrative features, and the support of existing software) have allowed the emergence of large-scale hosting platforms, such as those offered by Amazon and Rackspace, that customers can trust to securely isolate their hosted virtual machines from those of other tenants despite physical co-location on the same physical hardware.

Are hypervisors worthy of this degree of trust? Proponents of virtualization claim that the small trusted computing base (TCB) and narrow interfaces afforded by a hypervisor provide strong isolation between the software components that share a host. In fact, the TCB of a mature virtualization platform is *larger* than that of a conventional server operating system. Even Type-1 hypervisors, such as Xen [8] and Hyper-V [84], rely on a privileged OS to provide additional shared services, such as drivers for physical devices, device emulation, and administrative tools. While the external interfaces to these services broaden the attack surface exposed to customer VMs, the internal interfaces *between* components within that OS

are not as narrow or as carefully protected as those between components of the hypervisor itself. This large control VM is the "elephant in the room", often ignored in discussing the security of these systems.

While TCB size may not be a direct representation of risk, the shared control VM *is* a real liability for these systems. In Xen, for instance, this control VM houses a smorgasbord of functionality: device emulation and multiplexing, system boot, administrative toolstack, etc. Each of these services is presented to multiple customer VMs over different, service-specific interfaces (see Figure 2.1). As these services are all part of a single monolithic TCB, a compromise of any of them places the entire platform in danger.

The history of OS development shows us how to address the problem of a large TCB: break it into smaller pieces, isolate those pieces from each other, and reduce each one to the least privilege consistent with its task [149]. However, the history of OS deployment demonstrates that "secure by design" OSes often generate larger communities of readers than developers or users. In this vein, from-scratch hypervisors [136, 139, 145] have shown that particular security properties can be achieved by rearchitecting the platform, but they do not provide the rich set of features necessary for deployment in commercial hosting environments.

The work described in this chapter avoids this compromise: we address the monolithic TCB presented by the control VM *without* reducing functionality. Instead, we hold the features of a mature, deployed hypervisor as a baseline and harden the underlying TCB. Our approach is to incorporate stronger isolation for the existing components in the TCB, increasing our ability to control and reason about exposure to risk. While full functionality is necessary, it is not sufficient for commercial deployment. Our approach adds only a small amount of performance overhead compared to our starting point full-featured virtualization platform.

2.1.1 Contributions

The primary contribution of this chapter is to perform a component-based disaggregation of a mature, broadly deployed virtualization platform in a

manner that is practical to incorporate and maintain. Our work takes advantage of a number of well-established mechanisms that have been used to build secure and reliable systems: the componentization of microkernels, freshening of component state using microreboots [27], and the use of recovery boxes [6] to allow a small set of explicitly designated state to survive reboots. The insight in this work is that these techniques can be applied to an existing system along the boundaries that already exist between processes and interfaces in the control VM.

We describe the challenges of decomposing Xen's control VM into a set of nine classes of *service VMs* while maintaining functional, performance, and administrative parity. The resulting system, which we have named *Xoar*, demonstrates a number of interesting new capabilities that are not possible without disaggregation:

- **Disposable Bootstrap.** Booting the physical computer involves a great deal of complex, privileged code. Xoar isolates this functionality in special purpose service VMs and destroys these VMs before the system begins to serve users. Other Xoar components are microrebooted to known-good snapshots, allowing developers to reason about a specific software state that is ready to handle a service request.

- **Auditable Configurations.** As the dependencies between customer VMs and service VMs are explicit, Xoar is able to record a secure audit log of all configurations that the system has been placed in as configuration changes are made. We show that this log can be treated as a temporal database, enabling providers to issue forensic queries, such as asking for a list of VMs that depended on a known-vulnerable component.

- **Hardening of Critical Components.** While a core goal of our work has been to minimize the changes to source in order to make these techniques adoptable and maintainable, some critical components are worthy of additional attention. We identify *XenStore*, Xen's service for managing configuration state and inter-VM communication, as a

sensitive and long-running component that is central to the security of the system. We show how isolation and microreboots allow XenStore to be rearchitected in a manner whereby an attacker must be capable of performing a stepping-stone attack across two isolated components in order to compromise the service.

We believe that Xoar represents a real improvement to the security of these important systems, in a manner that is practical to incorporate today. After briefly describing our architecture, we present a detailed design and implementation. We end by discussing the security of the system and evaluate the associated performance costs.

2.2 TCBs, Trust, and Threats

This section describes the TCB of an enterprise virtualization platform and articulates our threat model. It concludes with a classification of relevant existing published vulnerabilities as an indication of threats that have been reported in these environments.

TCBs: Trust and Exposure: The TCB is classically defined as "the totality of protection mechanisms within a computer system—including hardware, firmware, and software—the combination of which is responsible for enforcing a security policy" [1]. In line with existing work on TCB reduction, we define the TCB of a subsystem S as "the set of components that S trusts not to violate the security of S" [69, 116].

Enterprise virtualization platforms, such as Xen, VMware ESX, and Hyper-V, are responsible for the isolation, scheduling, and memory management of guest VMs. Since the hypervisor runs at the highest privilege level, it forms, along with the hardware, part of the system's TCB.

Architecturally, these platforms rely on additional components. Device drivers and device emulation components manage and multiplex access to I/O hardware. Management toolstacks are required to actuate VMs running on the system. Further components provide virtual consoles, configuration state management, inter-VM communication, and so on. Commodity

virtualization platforms, such as the ones mentioned above, provide all of these components in a monolithic domain of trust, either directly within the hypervisor or within a single privileged virtual machine running on it. Figure 2.1 illustrates an example of this organization as implemented in Xen.

A compromise of any component in the TCB affords the attacker two benefits. First, they gain the privileges of that component, such as access to arbitrary regions of memory or control of hardware. Second, they can access its interfaces to other elements of the TCB, which allows them to attempt to inject malicious requests or responses over those interfaces.

Example Attack Vectors: We analyzed the CERT vulnerability database and VM-ware's list of security advisories, identifying a total of 44 reported vulnerabilities in Type-1 hypervisors.[1] Of the reported Xen vulnerabilities, 23 originated from within guest VMs, 11 of which were buffer overflows allowing arbitrary code execution with elevated privileges, while the other eight were denial-of-service attacks. Classifying by attack vector showed 14 vulnerabilities in the device emulation layer, with another two in the virtualized device layer. The remainder included five in management components and only two hypervisor exploits. 21 of the 23 attacks outlined above are against service components in the control VM.

Threat Model: We assume a well-managed and professionally administered virtualization platform that restricts access to both physical resources and privileged administrative interfaces. That is, we are not concerned with the violation of guest VM security by an administrator of the virtualization service. There are business imperatives that provide incentives for good behavior on the part of hosting administrators.

There is no alignment of incentives, however, for the guests of a hosting service to trust each other, and this forms the basis of our threat model. In

[1] There were a very large number of reports relating to Type-2 hypervisors, most of which assume the attacker has access to the host OS and compromises known OS vulnerabilities—for instance, using Windows exploits to compromise VMware Workstation. These attacks are not representative of our threat model and are excluded.

a multi-tenancy environment, since guests may be less than well administered and exposed to the Internet, it is prudent to assume that they may be malicious. Thus, the attacker in our model is a guest VM aiming to violate the security of another guest with whom it is sharing the underlying platform. This includes violating the data integrity or confidentiality of the target guest or exploiting the code of the guest.

While we assume that the hypervisor of the virtualization platform is trusted, we also assume that the code instantiating the functionality of the control VM *will* contain bugs that are a potential source of compromise. Note that in the case of a privileged monolithic control VM, a successful attack on any one of its many interfaces can lead to innumerable exploits against guest VMs. Rather than exploring techniques that might allow for the construction of a bug-free platform, our more pragmatic goal is to provide an architecture that isolates functional components in space and time so that an exploit of one component is not sufficient to mount a successful attack against another guest or the underlying platform.

2.3 Architecture Overview

Before explaining the design goals behind Xoar, it is worth providing a high-level overview of the components to help clarify the complexities of the control plane in a modern hypervisor and to establish some of the Xen-specific terminology that is used throughout the remainder of the chapter. While our implementation is based on Xen, other commercial Type-1 hypervisors, such as those offered by VMware and Microsoft, have sufficiently similar structures that we believe the approach presented in this chapter is applicable to them as well.

2.3.1 The Xen Platform

The Xen hypervisor relies on its control VM, Dom0, to provide a virtualized I/O path and host a system-wide registry and management toolstack.

Device Drivers: Xen delegates the control of PCI-based peripherals, such as network and disk controllers, to Dom0, which is responsible for exposing a set of abstract devices to guest VMs. These devices may either be virtualized, passed through, or emulated.

Virtualized devices are exposed to customer VMs using a "split driver" model [51]. A backend driver, having direct control of the hardware, exposes virtualized devices to frontend drivers in the guest VMs. Frontend and backend drivers communicate over a shared memory ring, with the backend multiplexing requests from several frontends onto the underlying hardware. Xen is involved only in enforcing access control for the shared memory and passing synchronization signals. Access Control Lists (ACLs) are stored in the form of *grant tables*, with permissions set by the owner of the memory.

Alternatively, Xen uses direct device assignment to allow VMs other than Dom0 to directly interface with passed-through hardware devices. Dom0 provides a virtual PCI bus, using a split driver, to proxy PCI configuration and interrupt assignment requests from the guest VM to the PCI bus controller. Device-specific operations are handled directly by the guest. Direct assignment can be used to move physical device drivers out of Dom0, in particular for PCI hardware that supports hardware-based IO virtualization (SR-IOV) [94].

Unmodified commodity OSes, on the other hand, expect to run on a standard hardware platform with a BIOS and specific hardware components that are supported with pre-packaged device drivers. In the absence of these components, the platform can be emulated entirely in software. In Xen, this device emulation layer is provided by a per-guest Qemu [13] instance, running either as a Dom0 process or in its own VM [154]. It has privileges to map any page of the guest's memory in order to emulate DMA operations.

XenStore: XenStore is a hierarchical key-value store that acts as a system-wide registry and naming service. It also provides a "watch" mechanism that notifies registered listeners of any modifications to particular keys in the store. Device drivers and the toolstack make use of this for inter-VM synchronization and device setup.

XenStore runs as a Dom0 process and communicates with other VMs via shared memory rings. Since it is required in the creation and boot-up of a VM, it relies on Dom0 privileges to access shared memory directly, rather than using grant tables.

Despite the simplicity of its interface with VMs, the complex, shared nature of XenStore makes it vulnerable to DoS attacks if a VM monopolizes its resources [34]. Because it is the central repository for configuration state in the system and virtually all components in the system depend on it, it is a critical component from a security perspective. Exploiting XenStore allows an attacker to deny service to the system as a whole and to perform most administrative operations, including starting and stopping VMs, and possibly abusing interfaces to gain access to guest memory or other guest VMs.

Other systems (including previous versions of Xen) have used a completely message-oriented approach, either as a point-to-point implementation or as a message bus. Having implemented all of these at various points in the past (and some of them more than once), our experience is that they are largely isomorphic with regard to complexity and decomposability.

Toolstack: The toolstack provides administrative functions for the management of VMs. It is responsible for creating, destroying, and managing the associated resources and privileges of VMs. Creating a VM requires Dom0 privileges to map guest memory to load a kernel or virtual BIOS and to set up initial communication channels with XenStore and the virtual console. In addition, the toolstack registers newly created guests with XenStore.

System Boot: In a traditional Xen system, the boot process is simple: the hypervisor creates Dom0 during boot-up, which proceeds to initialize hardware and bring up devices and their associated backend drivers. XenStore is started before any guest VM is created.

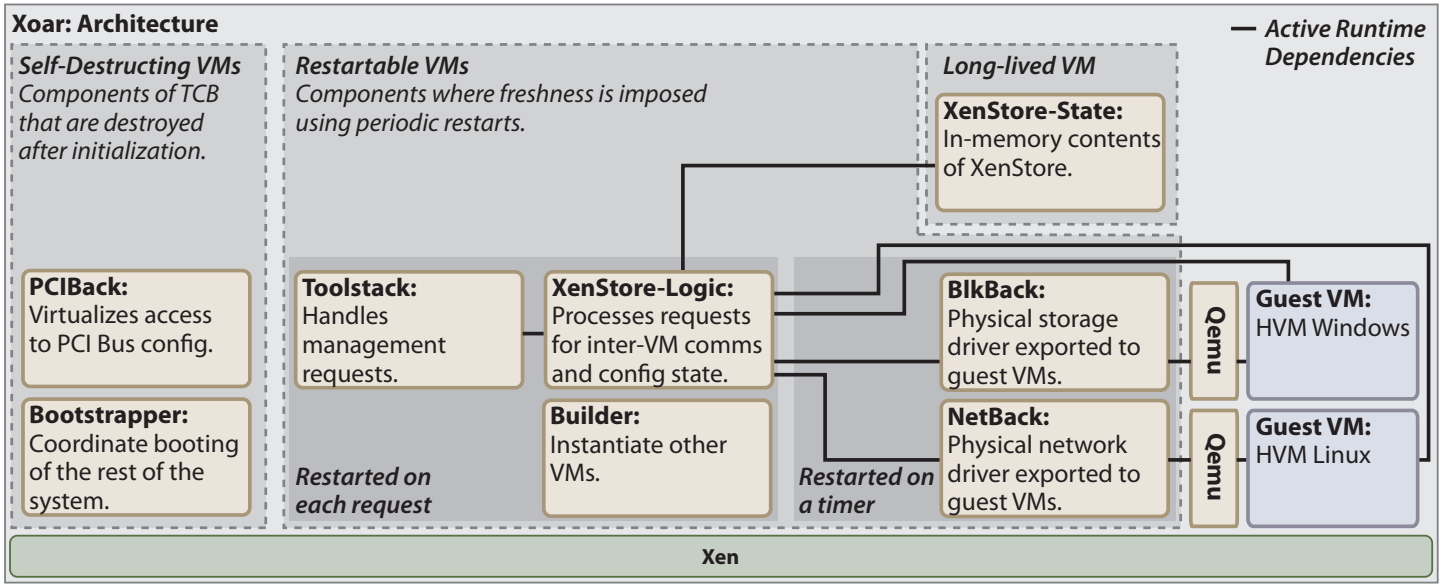

Figure 2.2: Architecture of Xoar. The figure above shows all the classes of service VMs along with the dependencies between them. For clarity, ephemeral dependencies (e.g., between the Builder and the VMs that it builds) are not shown. As suggested in the figure, a Qemu service VM is instantiated for the lifetime of each guest.

2.3.2 Xoar

Figure 2.2 shows the architecture of Xoar, and will be referred to throughout the remainder of this chapter. In Xoar, the functionality of Xen's control VM has been disaggregated into nine classes of service VMs, each of which contains a single-purpose piece of control logic that has been removed from the original monolithic control VM. As is the case with the monolithic TCB, some components may have multiple instances, each serving different client VMs.

That these individual components may be instantiated more than once is important, as it allows them to be used as flexible building blocks in the deployment of a Xoar-based system. Figure 2.2 shows a single instance of each component other than the QemuVM. Later in the chapter we will describe how multiple instances of these components, with differing resource and privilege assignments, can partition and otherwise harden the system as a whole.

From left to right, we begin with two start-of-day components that are closely tied to booting the hypervisor itself, *Bootstrapper* and *PCIBack*. These components bring up the physical platform and interrogate and configure hardware. In most cases this functionality is required only when booting the system and so these components are destroyed before any customer VMs are started. This is a useful property in that platform drivers and PCI discovery represent a large volume of complex code that can be removed prior to the system entering a state where it may be exposed to attacks.

While PCIBack is logically a start-of-day component, it is actually created after *XenStore* and *Builder*. XenStore is required to virtualize the PCI bus and the Builder is the only component capable of creating new VMs on the running system. PCIBack uses these components to create device driver VMs during PCI device enumeration by using `udev` rules [92].

Three components are responsible for presenting platform hardware that is not directly virtualized by Xen. *BlkBack* and *NetBack* expose virtualized disk and network interfaces and control the specific PCI devices that have been assigned to them. For every guest VM running an unmodified OS, there

is an associated *QemuVM* responsible for device emulation.

Once the platform is initialized, higher-level control facilities like the *Toolstacks* are created. The Toolstacks request the Builder to create guest VMs. As a control interface to the system, toolstacks are generally accessed over a private enterprise network, isolated from customer VM traffic.

As in Xen, a VM is described using a configuration file that is provided to the toolstack. This configuration provides runtime parameters such as memory and CPU allocations, and also device configurations to be provided to the VM. When a new VM is to be created, the toolstack parses this configuration file and writes the associated information into XenStore. Other components, such as driver VMs, have watches registered which are triggered by the build process, and configure connectivity between themselves and the new VM in response. While Xoar decomposes these components into isolated virtual machines, it leaves the interfaces between them unchanged; XenStore continues to be used to coordinate VM setup and tear down. The major difference is that privileges, both in terms of access to configuration state within XenStore and access to administrative operations in the hypervisor, are restricted to the specific service VMs that need them.

2.4 Design

In developing Xoar, we set out to maintain functional parity with the original system and complete transparency with existing management and VM interfaces, including legacy support, without incurring noticeable overhead. This section discusses the approach that Xoar takes, and the properties that were considered in selecting the granularity and boundaries of isolation.

Our design is motivated by these three goals:

1. **Reduce privilege** Each component of the system should have only the privileges essential to its purpose; interfaces exposed by a component, both to dependent VMs and to the rest of the system, should be the minimal set necessary. This confines any successful attack to the limited capabilities and interfaces of the exploited component.

2. **Reduce sharing** Sharing of components should be avoided wherever it is reasonable; whenever a component is shared between multiple dependent VMs, this sharing should be made explicit. This enables reasoning and policy enforcement regarding the exposure to risk introduced by depending on a shared component. It also allows administrators to securely log and audit system configurations and to understand exposure after a compromise has been detected.

3. **Reduce staleness** A component should only run for as long as it needs to perform its task; it should be restored to a known good state as frequently as practicable. This confines any successful attack to the limited execution time of the exploited component and reduces the execution state space that must be tested and evaluated for correctness.

To achieve these goals, we introduce an augmented version of the virtual machine abstraction: the *service VM*. Service VMs are the units of isolation that host the service components of the control VM. They differ from conventional virtual machines in that only service VMs can receive any extra privilege from the hypervisor or provide services to other VMs. They are also the only components that can be shared in the system, aside from the hypervisor itself.

Service VMs are entire virtual machines, capable of hosting full OSes and application stacks. Individual components of the control VM, which are generally either driver or application code, can be moved in their entirety out of the monolithic TCB and into a service VM. The hypervisor naturally assigns privilege at the granularity of the tasks these components perform. As such, there is little benefit, and considerable complexity, involved in finer-grained partitioning.

Components receiving heightened privilege and providing shared services are targets identified by the threat model discussed in Section 2.2. By explicitly binding their capabilities to a VM, Xoar is able to directly harden the riskiest portions of the system and provide service-specific enhancements for security. The remainder of this section discusses the design of

assign_pci_device (*PCI_domain*, *bus*, *slot*) **permit_hypercall** (*hypercall_id*) **allow_delegation** (*guest_id*)

Figure 2.3: Privilege Assignment API

Xoar with regard to each of these three goals.

2.4.1 Privilege: Fracture the Monolithic TCB

A service VM is designated as such using a `serviceVM` block in a VM config file. This block indicates that the VM should be treated as an isolated component and contains parameters that describe its capabilities. Figure 2.3 shows the API for the assignment of the three privilege-related properties that can be configured: direct hardware assignment, privileged hypercalls, and the ability to delegate privileges to other VMs on creation.

Direct hardware assignment is already supported by many x86 hypervisors, including Xen. Given a PCI domain, bus, and slot number, the hypervisor validates that the device is available to be assigned and is not already committed to another VM, then allows the VM to control the device directly.

Hypercall permissions allow a service VM access to some of the privileged functionality provided by the hypervisor. The explicit white-listing of hypercalls beyond the default set available to guest VMs allows for least-privilege configuration of individual service VMs. These permissions are translated directly into Flask [144], a flexible, fine-grained policy engine that assigns roles to different service VMs and restricts their ability to make privileged hypercalls, and installed into the hypervisor.

Access to resources is restricted by delegating service VMs to only those Toolstacks allowed to utilize those resources to support newly created VMs. Attempts to use undelegated service VMs are blocked by the hypervisor, enabling coarse-grained partitioning of resources. In the private cloud example presented at the end of this section, each user is assigned a private Toolstack, with delegated service VMs, and has exclusive access to the underlying hardware.

resource =	[*provider*, *parameters*, ***constraint_group***=*tag*]

Figure 2.4: Constraint Tagging API

2.4.2 Sharing: Manage Exposure

Isolating the collection of shared services in service VMs confines and restricts attacks and allows an explicit description of the relationships between components in the system. This provides a clear statement of configuration constraints to avoid exposure to risk and enables mechanisms to reason about the severity and consequences of compromises after they occur.

Configuration Constraints: A guest can provide constraints on the service VMs that it is willing to use. At present, a single constraint is allowed, as shown in Figure 2.4. The `constraint_group` parameter provides an optional user-specified tag and may be appended to any line specifying a shared service in the VM's configuration. Xoar ensures that no two VMs specifying different constraint groups ever share the same service VM.

Effectively, this constraint is a user-specified coloring that prevents sharing. By specifying a tag on all of the devices of their hosted VMs, users can insist that they be placed in configurations where they only share service VMs with guest VMs that they control.

Secure Audit: Xoar borrows techniques from past forensics systems such as Taser [54]. The coarse-grained isolation and explicit dependencies provided by service VMs makes these auditing approaches easier to apply. Whenever the platform performs a guest-related configuration change (e.g., the creation, deletion, pausing, or unpausing of a VM), Xoar logs the resulting dependencies to an off-host, append-only database over a secure channel. Currently, we use the temporal extension for Postgres.

Two simple examples show the benefit of this approach. First, the top query in Figure 2.5 determines which customers could be affected by the compromise of a service VM by enumerating VMs that relied on that particular service VM at any point during the compromise. Second, providers

```
SELECT e1, e2 FROM log e1, log e2 WHERE
  e1.name = e2.name AND
  e1.action = 'create' AND
  e2.action = 'destroy' AND
  e1.dependency = 'NameOfCompromisedNetBack' AND
  overlaps(period_intersect(e1.time, e2.time),
           compromise_period);


SELECT e1.name FROM log e1 WHERE
  e1.dependency = 'NetBack' AND
  e1.dependency_version = vulnerable_version;
```

Figure 2.5: Temporal queries which search for guest VMs that depended on a service VM that was compromised (top) or vulnerable (bottom).

frequently roll out new versions of OS kernels and in the event that a vulnerability is discovered in a specific release of a service VM after the fact, the audit log can be used to identify all guest VMs that were serviced by it.

2.4.3 Staleness: Protect VMs in Time

The final feature of service VMs is a facility to defend the *temporal* attack surface, preserving the freshness of execution state through the use of periodic restarts. This approach takes advantage of the observation from work on microreboots and "crash-only software" [27] that it is generally easier to reason about a program's correctness at the start of execution rather than over long periods of time.

Microreboots: Virtual machines naturally support a notion of rebooting that can be used to reset them to a known-good state. Further, many of the existing interfaces to control VM-based services already contain logic to reestablish connections, used when migrating a running VM from one physical host to another. There are two major challenges associated with microreboots. First, full system restarts are slow and significantly reduce performance, especially of components on a data path such as device drivers. Second, not all state associated with a service can be discarded since useful side-effects

Calls from within the service VM:
vm_snapshot ()
recoverybox_balloc (*size*)

VM configuration for restart policy:
restart_policy ([*(timer | event), parameters*])

Figure 2.6: Microreboot API

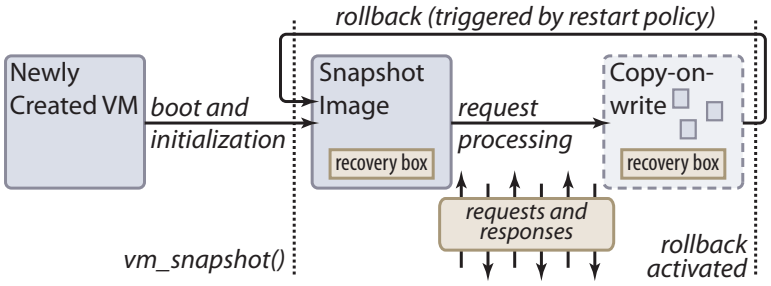

Figure 2.7: Rolling back to a known-good snapshot allows efficient microreboots of components.

that have occurred during that execution will also be lost.

Snapshot and Rollback: Instead of fully restarting a component, it is snapshotted just after it has booted and been initialized, but before it has communicated with any other service or guest VM. The service VM is modified to explicitly snapshot itself at the time that it is ready to service requests (typically at the start of an event loop) using the API shown in Figure 2.6. Figure 2.7 illustrates the snapshot/rollback cycle. By snapshotting before any requests are served over offered interfaces, we ensure that the image is fresh. A complementary extension would be to measure and attest snapshot-based images, possibly even preparing them as part of a distribution and avoiding the boot process entirely.

We enable lightweight snapshots by using a hypervisor-based copy-on-write mechanism to trap and preserve any pages that are about to be modified. When rolling back, only these pages and the virtual CPU state need be restored, resulting in very fast restart times—in our implementation, be-

tween 4 and 25 ms, depending on the workload.

Restart Policy: While it is obvious when to take the snapshot of a component, it is less clear when that component should be rolled back. Intuitively, it should be as frequently as possible. However, even though rollbacks are quick, the more frequently a component is restarted, the less time it has available to offer a useful service. Xoar specifies rollback policy in the service VM's configuration file and we currently offer two policies: notification-based and timer-based. Restart policy is associated with the VM when it is instantiated and is tracked and enforced by the hypervisor.

In our notification-based policy, the hypervisor interposes on message notifications *leaving* the service VM as an indication that a request transaction has completed, triggering a restart. For low-frequency, synchronous communication channels (e.g., those that access XenStore), this method isolates individual transactions and resets the service to a fresh state at the end of every processed request. In other words, every single request is processed by a fresh version of the service VM.[2]

The overhead of imposing a restart on every request would be too high for higher-throughput, concurrent channels, such as NetBack and BlkBack. For these service VMs, the hypervisor provides a periodic restart timer that triggers restarts at a configurable frequency.

Maintaining State: Frequent restarts suffer from the exact symptom that they seek to avoid: the establishment of long-lived state. In rolling back a service VM, any state that it introduces is lost. This makes it particularly hard to build services that depend on keeping in-memory state, such as configuration registries, and services that need to track open connections.

We address this issue by providing service VMs with the ability to allocate a *recovery box* [6]. Originally proposed as a technique for high availability, this is a block of memory that persists across restarts. Service VM code is

[2]This mechanism leaves open the possibility that an exploited service VM might not send the event that triggers the rollback. To cover this attack vector, the hypervisor maintains a watchdog timer for each notification-based service VM. If a timer goes off, the VM is rolled back; if the restart is triggered normally, the timer is reset.

modified to store any long-lived state in one of these allocations and to check and restore from it immediately after a snapshot call. Memory allocated using this technique is exempted from copy-on-write.

Maintaining state across restarts presents an obvious attack vector—a malicious user can attempt to corrupt the state that is reloaded after every rollback to repeatedly trigger the exploit and compromise the system. To address this, the service treats the recovery box as an untrusted input and audits its contents after the rollback. Xen also tracks the memory pages in the allocation and forcibly marks all virtual addresses associated with them as non-executable.

Driver VMs, like NetBack and BlkBack, automatically renegotiate both device state and frontend connections in cases of failures or restarts, allowing them to discard all state at every restart. In these performance-critical components, however, any downtime significantly affects the throughput of guests. This downtime can be reduced by caching a very small amount of device and frontend state in a recovery box. The desired balance between security and performance can be chosen, as discussed in Section 2.7.2.

Components like XenStore, on the other hand, maintain a large amount of long-lived state for other components in the system. In such cases, this state can be removed from the service VM altogether and placed in a separate "state" VM that is accessible through a special-purpose interface. In Xoar, only XenStore, because of its central role in the correctness and security of the system, is refactored in this way. Only the processing and logic remain in the original service VM, making it amenable to rollbacks.

Per-request rollbacks force the attacker to inject exploit code into the state and have it triggered by another VM's interaction with XenStore. However, in the absence of further exploits, access control and guest ID authentication prevent the injection of such exploit code into sections of the state not owned by the attacking guest (see Section 2.5.2). Thus, an attack originating from a guest VM through XenStore requires an exploit of more than one service VM.

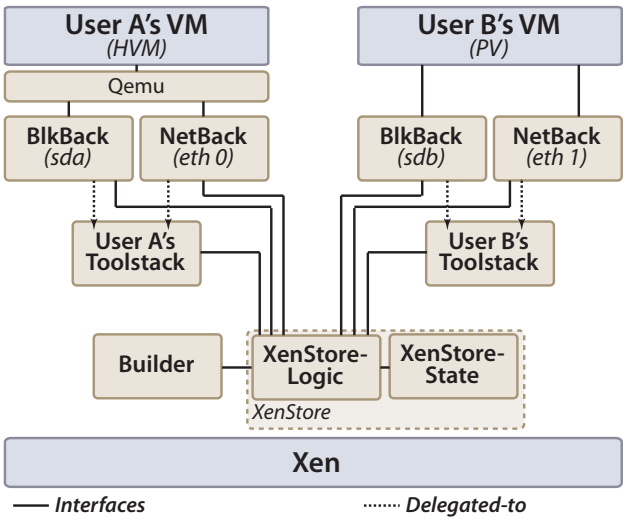

Figure 2.8: Partitioned configuration: In the configuration above, users A and B use isolated hardware and toolstacks and share interfaces only with XenStore and Xen itself.

2.4.4 Deployment Scenarios

Public clouds, like Amazon Web Services, tightly pack many VMs on a single physical machine, controlled by a single toolstack. Partitioning the platform into service VMs, which can be judiciously restarted, limits the risks of sharing resources among potentially vulnerable and exposed VMs. Furthermore, dynamically restarting service VMs allows for in-place upgrades, reducing the window of exposure in the face of a newly discovered vulnerability. Finally, in the case of compromise, secure audit facilities allow administrators to reason, after the fact, about exposures that may have taken place.

Our design supports greater degrees of resource partitioning than this. Figure 2.8 shows a more conservative configuration, in which each user is assigned separate, dedicated hardware resources within the physical host and a personal collection of service VMs to manage them. Users manage their own service VMs and the device drivers using a private Toolstack with resource service VMs delegated solely to it.

Component	P	Lifetime	OS	Parent	Depends On	Functionality
Bootstrapper	Y	Boot Up	nanOS	Xen	-	Instantiate boot service VMs
XenStore	N	Forever (R)	miniOS	Bootstrapper	-	System configuration registry
Console	N	Forever	Linux	Bootstrapper	XenStore	Expose physical console as virtual consoles to VMs
Builder	Y	Forever (R)	nanOS	Bootstrapper	XenStore	Instantiate non-boot VMs
PCIBack	Y	Boot Up	Linux	Bootstrapper	XenStore Builder Console	Initialize hardware and PCI bus, pass through PCI devices, and expose virtual PCI config space
NetBack	N	Forever (R)	Linux	PCIBack	XenStore Console	Expose physical network device as virtual devices to VMs
BlkBack	N	Forever (R)	Linux	PCIBack	XenStore Console	Expose physical block device as virtual devices to VMs
Toolstack	N	Forever (R)	Linux	Bootstrapper	XenStore Builder Console	Admin toolstack to manage VMs
QemuVM	N	Guest VM	miniOS	Toolstack	XenStore NetBack BlkBack	Device emulation for a single guest VM

Table 2.1: Components of Xoar. The "P" column indicates if the component is privileged. An "(R)" in the lifetime column indicates that the component can be restarted. Console is only mentioned for the sake of completeness. Since enterprise deployments typically disable console access, it is not part of the overall architecture.

2.5 Implementation

This section explains how the design described in Section 2.4 was implemented on the Xen platform. It begins with a brief discussion of how component boundaries were selected in fracturing the control VM and then describes implementation details and challenges faced during the development of Xoar.

2.5.1 Xoar Components

The division of service VMs in Xoar conforms to the design goals of Section 2.4; we reduce components into minimal, loosely coupled units of functionality, while obeying the principle of least privilege. As self-contained units, they have a low degree of sharing and inter-VM communication (IVC), and can be restarted independently. Existing software and interfaces are reused to aid development and ease future maintenance. Table 2.1 augments Figure 2.2 by describing the classes of service VMs in our decomposition of Dom0. While it is not the only possible decomposition, it satisfies our design goals without requiring an extensive re-engineering of Xen.

Virtualized devices mimic physical resources in an attempt to offer a familiar abstraction to guest VMs, making them ideal service VMs. Despite the lack of toolstack support, Xen has architectural support for driver VMs, reducing the development effort significantly. PCIBack virtualizes the physical PCI bus, while NetBack and BlkBack are driver VMs, exposing the required device backends for guest VMs. Further division, like separating device setup from the data path, yields no isolation benefits, since both components need to be shared simultaneously. This would also add a significant amount of IVC, conflicting with our design goals, and would require extensive modifications. Similarly, the serial controller is represented by a service VM that virtualizes the console for other VMs. Further details about virtualizing these hardware devices are discussed in Section 2.5.3 and Section 2.5.4.

Different aspects of the VM creation process require differing sets of privileges; placing them in the same service VM violates our goal of reducing privilege. These operations can largely be divided into two groups—

those that need access to the guest's memory to set up the kernel, etc., and those that require access to XenStore to write entries necessary for the guest. Breaking this functionality apart along the lines of least privilege yields the Builder, a privileged service VM responsible for the hypervisor and guest memory operations, and the Toolstack, a service VM containing the management toolstack. While the Builder could be further divided into components for sub-operations, like loading the kernel image, setting up the page tables, etc., these would all need to run at the same privilege level and would incur high synchronization costs. The Builder responds to build requests issued by the Toolstack via XenStore. Once building is complete, the Toolstack communicates with XenStore to perform the rest of the configuration and setup process.

2.5.2 XenStore

Our refactoring of XenStore is the most significant implementation change that was applied to any of the existing components in Xen (and took the largest amount of effort). We began by breaking XenStore into two independent service VMs: XenStore-Logic, which contains the transactional logic and connection management code, and XenStore-State, which contains the actual contents of the store. This division allows restarts to be applied to request-handling code on a per-request basis, ensuring that exploits are constrained in duration to a single request. XenStore-State is a simple key-value store and is the only long-lived VM in Xoar.

Unfortunately, partitioning and per-request restarts are insufficient to ensure the security of XenStore. As XenStore-Logic is responsible for enforcing access control based on permissions in the store itself, a compromise of that VM may allow for arbitrary accesses to the contents of the store. We addressed this problem with two techniques. First, access control checks are moved into a small monitor module in XenStore-State; a compromise of XenStore-Logic is now limited to valid changes according to existing permissions in the store. Second, we establish the authenticity of accesses made by XenStore-Logic by having it declare the identity of the VM that it is about

to service *before* reading the actual request. This approach effectively drops privilege to that of a single VM before exposing XenStore-Logic to any potentially malicious request, and makes the identity of the request made to XenStore-State unforgeable. The monitor refuses any request to change the current VM until the request has been completed, and an attempt to do so results in a restart of XenStore-Logic.

The monitor code could potentially be further disaggregated from XenStore-State and also restarted on a per-request basis. Our current implementation requires an attacker to compromise both XenStore-Logic and the monitor code in XenStore-State in succession, within the context of a single request, in order to make an unauthorized access to the store. Decoupling the monitor from XenStore-State would add limited extra benefit, for instance possibly easing static analysis of the two components, and still allow a successful attacker to make arbitrary changes in the event of the two successive compromises; therefore we have left the system is it stands.

2.5.3 PCI: A Shared Bus

PCIBack controls the PCI bus and manages interrupt routing for peripheral devices. Although driver VMs have direct access to the peripherals themselves, the shared nature of the PCI configuration space requires a single component to multiplex all accesses to it. This space is used during device initialization, after which there is no further communication with PCIBack. We remove PCIBack from the TCB entirely after boot by destroying it, reducing the number of shared components in the system.

Hardware virtualization techniques like SR-IOV [94] allow the creation of virtualized devices, where the multiplexing is performed in hardware, obviating the need for driver VMs. However, provisioning new virtual devices on the fly requires a persistent service VM to assign interrupts and multiplex accesses to the PCI configuration space. Ironically, although appearing to reduce the amount of sharing in the system, such techniques may increase the number of shared, trusted components.

2.5.4 Driver VMs: NetBack and BlkBack

Driver VMs, like NetBack and BlkBack, use direct device assignment to directly access PCI peripherals like NICs and disk controllers, and rely on existing driver support in Linux to interface with the hardware. Each NetBack or BlkBack virtualizes exactly one network or block controller, hosting the relevant device driver and virtualized backend driver. The Toolstack links a driver VM delegated to it to a guest VM by writing the appropriate frontend and backend XenStore entries during the creation of the guest, after which the guest and backend communicate directly using shared memory rings, without any further participation by XenStore.

Separating BlkBack from the Toolstack caused some problems as the existing management tools mount disk-based VM images as loopback devices with `blktap`, for use by the backend driver. After splitting BlkBack from the Toolstack, the disk images need to be created and mounted in BlkBack. Therefore, in Xoar, BlkBack runs a lightweight daemon that proxies requests from the Toolstack.

2.5.5 Efficient Microreboots

As described in Section 2.4.3, our snapshot mechanism copies dirty memory pages, i.e., pages that have been modified since the last good snapshot, as a service VM executes and restores the original contents of these pages during rollback, requiring a page allocation and deallocation and two copy operations for every dirtied page. Since many of the pages being modified are the same across several iterations, rather than deallocating the master copies of these pages after rollback, we retain them across runs, obviating the need for allocation, deallocation, and one copy operation when the same page is dirtied. However, this introduces a new problem: if a page is dirtied just once, its copy will reside in memory forever. This could result in memory being wasted storing copies of pages that are not actively required.

To address this concern, we introduced a "decay" value to the pages stored in the snapshot image. When a page is first dirtied after a rollback, its decay value is incremented by two, towards a maximum value. On rollback,

each page's decay value is decremented. When this count reaches zero, the page is released.

2.5.6 Deprivileging Administrative Tools

XenStore and the Console require Dom0-like privileges to forcibly map shared memory, since they are required before the guest VM can set up its grant table mappings. To avoid this, Xoar's Builder creates grant table entries for this shared memory in each new VM, allowing these tools to use grant tables and function without any special privileges.

The Builder assigns VM management privileges to each Toolstack for the VMs that it requests to be built. A Toolstack can only manage these VMs, and an attempt to manage any others is blocked by the hypervisor. Similarly, it can use only those service VMs that have been delegated to it. An attempt to use an undelegated service VM, for example a NetBack, for a new guest VM will fail. Restricting privileges this way allows for the creation of several Toolstack instances that run simultaneously. Different users, each with a private Toolstack, are able to partition their physical resources and manage their own VMs, while still guaranteeing strong isolation from VMs belonging to other users.

2.5.7 Developing with Minimal OSes

Bootstrapper and Builder are built on top of nanOS, a small, single-threaded, lightweight kernel explicitly designed to have the minimum functionality needed for VM creation. The small size and simplicity of these components leave them well within the realm of static analysis techniques, which could be used to verify their correctness. XenStore, on the other hand, demands more from its operating environment, and so is built on top of miniOS, a richer OS distributed with Xen.

Determining the correct size of OS to use is hard, with a fundamental tension between functionality and ease of use. Keeping nanOS so rigidly simple introduces a set of development challenges, especially in cases involving IVC. However, since these components have such high privilege, we

felt that the improved security gained from reduced complexity is a worthwhile trade-off.

2.5.8 Implicit Assumptions about Dom0

The design of Xen does not mandate that all service components live in Dom0, however several components, including the hypervisor, implicitly hard-code the assumption that they do. A panoply of access control checks compare the values of domain IDs to the integer literal '0', the ID for Dom0. Many tools assume that they are running co-located with the driver backends and various paths in XenStore are hard-coded to be under Dom0's tree The toolstack expects to be able to manipulate the files that contain VM disk images, which is solved by proxying requests, as discussed in Section 2.5.4. The hypervisor assumes Dom0 has control of the hardware and configures signal delivery and MMIO and I/O-port privileges for access to the console and peripherals to Dom0. In Xoar, these need to be mapped to the correct VMs, with Console requiring the signals and I/O-port access for the console and PCIBack requiring the MMIO and remaining I/O-port privileges, along with access to the PCI bus.

2.6 Security Evaluation

Systems security is notoriously challenging to evaluate, and Xoar's proves no different. In an attempt to demonstrate the improvement to the state of security for commodity hypervisors, this section will consider a number of factors. First, we will evaluate the reduction in the size of the trusted computing base; this is an approach that we do not feel is particularly indicative of the security of a system, but has been used by a considerable amount of previous work and does provide some insight into the complexity of the system as a whole. Second, we consider how the attack surface presented by the control VM changes in terms of isolation, sharing, and per-component privilege in an effort to evaluate the exposure to risk in Xoar compared to other systems. Finally, we consider how well Xoar handles the existing published vulnerabilities first described in Section 2.2.

Much of this evaluation is necessarily qualitative: while we have taken efforts to evaluate against published vulnerabilities, virtualization on modern servers is still a sufficiently new technology with few disclosed vulnerabilities. Our sense is that these vulnerabilities may not be representative of the full range of potential attacks.

In evaluating Xoar's security, we attempt to characterize it from an attacker's perspective. One notable feature of Xoar is that for an adversary to violate our security claim, more than one service VM must have a vulnerability, and a successful exploit must be able to perform a stepping-stone attack. We will discuss why this is true, and characterize the nature of attacks that are still possible.

2.6.1 Reduced TCB

The Bootstrapper, PCIBack, and Builder service VMs are the most privileged components, with the ability to arbitrarily modify guest memory and control and assign the underlying hardware. These privileges necessarily make them part of the TCB, as a compromise of any one of these components would render the entire system vulnerable. Both Bootstrapper and PCIBack are destroyed after system initialization is complete, effectively leaving Builder as the only service VM in the TCB. As a result, the TCB is reduced from Linux's 7.6 million lines of code to Builder's 13,500 lines of code, both on top of the hypervisor's 280,000 lines of code.[3]

2.6.2 Attack Surface

Monolithic virtualization platforms like Xen execute service components in a single trust domain, with every component running at the highest privilege level. As a result, the security of the entire system is defined by that of the weakest component, and a compromise of any component gives an attacker full control of the system.

Disaggregating service components into their own VMs not only provides

[3] All lines of code were measured using David Wheeler's SLOCCount from `http://www.dwheeler.com/sloccount/`

Permission	Bootstrapper	PCIBack	Builder	Toolstack	BlkBack	NetBack
Arbitrarily access memory	X		X			
Access and virtualize PCI devices		X				
Create VMs	X		X			
Manage VMs	X		X	X		
Manage assigned devices					X	X

Table 2.2: Functionality available to the service VMs in Xoar. Components with access to no privileged hypercalls are not shown. In Xen, Dom0 possesses all of these functionalities.

strong isolation boundaries, it also allows privileges to be assigned on a per-component basis, reducing the effect a compromised service VM has on the entire system. Table 2.2 shows the privileges granted to each service VM, which corresponds to the amount of access that an attacker would have on successfully exploiting it.

Attacks originating from guest VMs can exploit vulnerabilities in the interfaces to NetBack, BlkBack, or XenStore (see Table 2.3). An attacker breaking into a driver VM gains access only to the degree that other VMs trust that device. Exploiting NetBack might allow for intercepting another VM's network traffic, but not access to arbitrary regions of its memory. On hosts with enough hardware, resources can be partitioned so that no two guests share a driver VM.

Where components reuse the same code, a single vulnerability could be sufficient to compromise them all. Service VMs like NetBack, BlkBack, Console, and Toolstack run the same core Linux kernel, with specific driver modules loaded only in the relevant component. As a result, vulnerabilities in the exposed interfaces are local to the associated service VM, but vulnera-

Component	Shared Interfaces
XenStore-Logic	XenStore-State, Console, Builder, PCIBack, NetBack, BlkBack, Guest
XenStore-State	XenStore-Logic
Console	XenStore-Logic
Builder	XenStore-Logic
PCIBack	XenStore-Logic, NetBack, BlkBack
NetBack	XenStore-Logic, PCIBack, Guest
BlkBack	XenStore-Logic, PCIBack, Guest
Toolstack	XenStore-Logic
Guest VM	XenStore-Logic, NetBack, BlkBack

Table 2.3: Interfaces shared between service VMs

bilities in the underlying framework and libraries may be present in multiple components. For better code diversity, service VMs could use a combination of Linux, FreeBSD, OpenSolaris, and other suitable OSes.

Highly privileged components like the Builder have very narrow interfaces and cannot be compromised without exploiting vulnerabilities in multiple components, at least one of which is XenStore. Along with the central role it plays in state maintenance and synchronization, this access to Builder makes XenStore an attractive target. Compromising XenStore-Logic may allow an attacking guest to store exploit code in XenStore-State, which, when restoring state after a restart, re-compromises XenStore-Logic. The monitoring code described in Section 2.5.2, however, prevents this malicious state from being restored when serving requests from any other guest VM, ensuring that they interact with a clean copy of XenStore.

2.6.3 Vulnerability Mitigation

With a majority of the disclosed vulnerabilities against Xen involving privilege escalation against components in Dom0, Xoar proves to be successful in containing all but two of them. Table 2.4 taxonomizes the vulnerabilities

Component	Arbitrary Code Exec.	DoS	File System Access
Hypervisor	0 / 1	0 / 1	0 / 0
Device Emulation	8 / 8	3 / 3	3 / 3
Virtualized Drivers	1 / 1	1 / 1	0 / 0
XenStore	0 / 0	1 / 1	0 / 0
Toolstack	1 / 1	2 / 2	1 / 1

Table 2.4: Vulnerabilities mitigated in Xoar. The numbers represent total mitigated over total identified.

discussed in Section 2.2 based on the vulnerable component and the type of vulnerability, along with the number that are successfully mitigated in Xoar.

The 14 device emulation attacks are completely mitigated, as the device emulation service VM has no rights over any VM except the one the attacker came from. The two attacks on the virtualized device layer and the three attacks against the toolstack affect only those VMs that shared the same BlkBack, NetBack, and Toolstack components. The vulnerability present in XenStore did not exist in our custom version. Since Xoar does not modify the hypervisor, the two hypervisor vulnerabilities remain equally exploitable.

One of the vulnerabilities in the virtualized drivers is against the block device interface and causes an infinite loop, resulting in a denial of service. Periodically restarting BlkBack forces the attacker to continuously recompromise the system. Since requests from different guests are serviced on every restart, the device would continue functioning with low bandwidth, until a patch could be applied to prevent further compromises.

2.7 Performance Evaluation

The performance of Xoar is evaluated against a stock Xen Dom0 in terms of memory overhead, I/O throughput, and overall system performance. Each service VM in Xoar runs with a single virtual CPU; in stock Xen Dom0 runs with 2 virtual CPUs, the configuration used in the commercial XenServer [32] platform. All figures are the average of three runs, with 95% confidence intervals shown where appropriate.

Component	Memory	Component	Memory
XenStore-Logic	32 MB	XenStore-State	32 MB
Console	128 MB	PCIBack	256 MB
NetBack	128 MB	BlkBack	128 MB
Builder	64 MB	Toolstack	128 MB

Table 2.5: Memory requirements of individual components

Our test system was a Dell Precision T3500 server, with a quad-core 2.67 GHz Intel Xeon W3520 processor, 4 GB of RAM, a Tigon 3 Gigabit Ethernet card, and an Intel 82801JIR SATA controller with a Western Digital WD3200AAKS-75L9A0 320 GB 7200 RPM disk. VMX, EPT, and IOMMU virtualization are enabled. We use Xen 4.1.0 and Linux 2.6.31[4] pvops kernels for the tests. Identical guests running an Ubuntu 10.04 system, configured with two VCPUs, 1 GB of RAM and a 15 GB virtual disk are used on both systems. For network tests, the system is connected directly to another system with an Intel 82567LF-2 Gigabit network controller.

2.7.1 Memory Overhead

Table 2.5 shows the memory requirements of each of the components in Xoar. Systems with multiple network or disk controllers can have several instances of NetBack and BlkBack. Also, since users can select the service VMs to run, there is no single figure for total memory consumption. In commercial hosting solutions, console access is largely absent rendering the Console redundant. Similarly, PCIBack can be destroyed after boot. As a result, the memory requirements range from 512 MB to 896 MB, assuming a single network and block controller, representing a saving of 30% to an overhead of 20% on the default 750 MB Dom0 configuration used by XenServer. All performance tests compare a complete configuration of Xoar with a standard Dom0 Xen configuration.

[4]Hardware issues forced us to use a 2.6.32 kernel for some of the components.

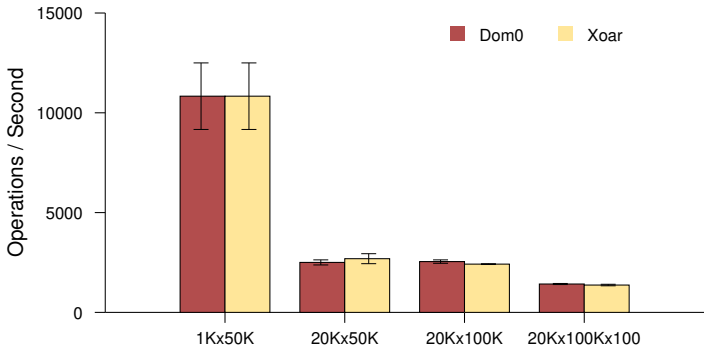

Figure 2.9: Disk performance using Postmark (higher is better). The x-axis denotes (files x transactions x subdirectories).

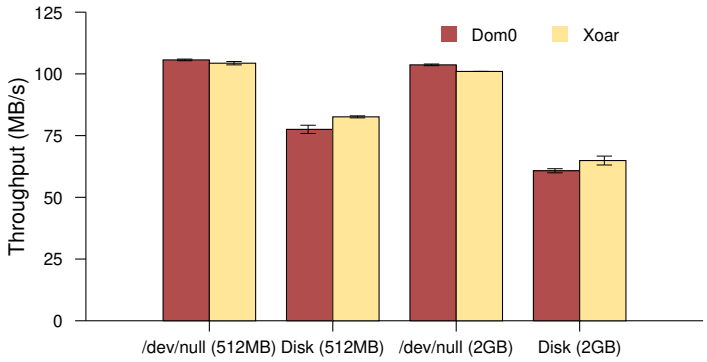

Figure 2.10: Network performance with `wget` (higher is better)

2.7.2 I/O performance

Disk performance is tested using Postmark, with VMs' virtual disks backed by files on a local disk. Figure 2.9 shows the results of these tests with different configuration parameters.

Network performance is tested by fetching a 512 MB and a 2 GB file across a gigabit LAN using `wget`, and writing it either to disk, or to `/dev/null` (to eliminate performance artifacts due to disk performance). The results are shown in Figure 2.10.

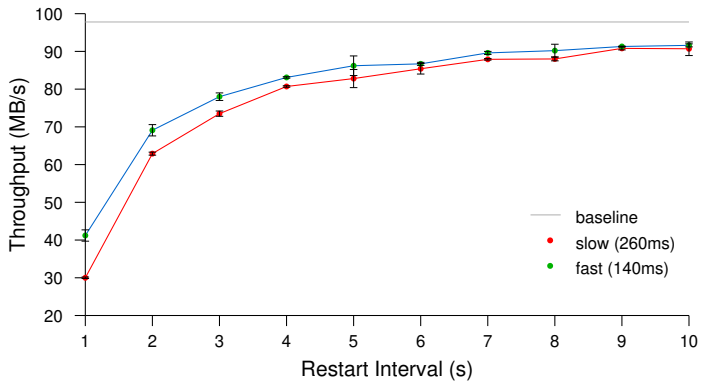

Figure 2.11: `wget` throughput while restarting NetBack at different time intervals

Overall, disk throughput is more or less unchanged, and network throughput is down by 1–2.5%. The combined throughput of data coming from the network onto the disk *increases* by 6.5%; we believe this is caused by the performance isolation of running the disk and network drivers in separate VMs.

To measure the effect of microrebooting driver VMs, we ran the 2 GB `wget` to `/dev/null` while restarting NetBack at intervals between 1 and 10 seconds. Two different optimizations for fast microreboots are shown.

In the first (marked as "slow" in Figure 2.11), the device hardware state is left untouched during reboots; in the second ("fast"), some configuration data that would normally be renegotiated via XenStore is persisted. In "slow" restarts the device downtime is around 260 ms, measuring from when the device is suspended to when it responds to network traffic again. The optimizations used in the "fast" restart reduce this downtime to around 140 ms.

Resetting every 10 seconds causes an 8% drop in throughput, as `wget`'s TCP connections respond to the breaks in connectivity. Reducing the interval to one second gives a 58% drop. Increasing it beyond 10 seconds makes very little difference to throughput. The faster recovery gives a noticeable benefit

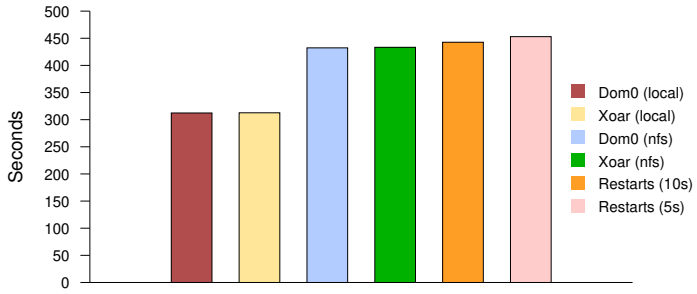

Figure 2.12: Linux kernel build run on Dom0 and Xoar, locally, over NFS and over NFS with NetBack restarts.

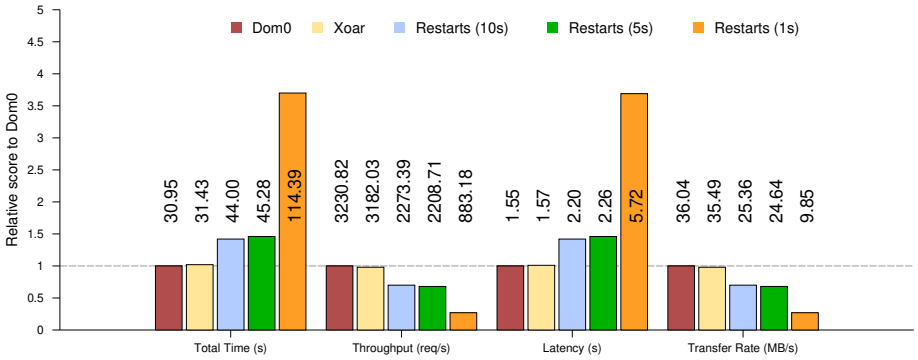

Figure 2.13: Apache Benchmark run on Dom0, Xoar, and Xoar with network driver restarts at 10s, 5s, and 1s.

for very frequent reboots but is worth less than 1% for 10-second reboots.

2.7.3 Real-world Benchmarks

Figure 2.12 compares the time taken to build a Linux kernel, both in stock Xen and Xoar, off a local ext3 volume as well as an NFS mount. The overhead added by Xoar is much less than 1%.

The *Apache Benchmark* is used to gauge the performance of an Apache web server serving a 10 KB static webpage 100,000 times to five simultaneous clients. Figure 2.13 shows the results of this test against Dom0, Xoar, and Xoar with network driver restarts at 10, 5, and 1 second intervals. Per-

formance decreases non-uniformly with the frequency of the restarts: an increase in restart interval from 5 to 10 seconds yields barely any performance improvements, while changing the interval from 5 seconds to 1 second introduces a significant performance loss.

Dropped packets and network timeouts cause a small number of requests to experience very long completion times; for example, for Dom0 and Xoar, the longest packet took only 8–9 ms, but with restarts, the values range from 3000 ms (at 5 and 10 seconds) to 7000 ms (at 1 second). As a result, the longest request interval is not shown in the figure.

Overall, the overhead of disaggregation is quite low. This is largely because driver VMs do not lengthen the data path between guests and the hardware: the guest VM communicates with NetBack or BlkBack, which drives the hardware. While the overhead of driver restarts is noticeable, as intermittent outages lead to TCP backoff, it can be tuned by the administrator to best match the desired combination of security and performance.

2.8 Related Work

With the widespread use of VMs, the security of hypervisors has been studied extensively and several attempts have been made to address the problem of securing the TCB. This section looks at some of these techniques in the context of our functional requirements.

Build a Smaller Hypervisor: SecVisor [136] and BitVisor [139] are examples of tiny hypervisors, built with TCB size as a primary concern, that use the interposition capabilities of hypervisors to retrofit security features for commodity OSes. While significantly reducing the TCB of the system, they do not share the multi-tenancy goals of commodity hypervisors and are unsuitable for such environments.

Microkernel-based architectures like KeyKOS [64] and EROS [137], its x86-based successor, are motivated similarly to Xoar and allow mutually untrusting users to securely share a system. Our Builder closely resembles the *factory* in KeyKOS. While multiple, isolated, independently administered

UNIX instances, rather like VMs, can be hosted on EROS, this requires modifications to the environment and arbitrary OSes cannot be hosted. More recently, NOVA [145] uses a similar architecture and explicitly partitions the TCB into several user-level processes within the hypervisor. Although capable of running multiple unmodified OSes concurrently, the removal of the control VM and requirement for NOVA-specific drivers sacrifice hardware support for TCB size. Also, it is far from complete: it cannot run Windows guests and has limited toolstack support.

NoHype [87] advocates removing the hypervisor altogether, using static partitioning of CPUs, memory, and peripherals among VMs. This would allow a host to be shared by multiple operating systems, but with none of the other benefits of virtualization. In particular, the virtualization layer could no longer be used for interposition, which is necessary for live migration [33], memory sharing and compression [62, 110], and security enhancements [31, 43, 100, 159].

Harden the Components of the TCB: The security of individual components of the TCB can be improved using a combination of improved code quality and access control checks to restrict the privileges of these components. Xen's XAPI toolstack is written in OCaml and benefits from the robustness that a statically typed, functional language provides [135]. Xen and Linux both have mechanisms to enforce fine-grained security policies [102, 133]. While useful, these techniques do not address the underlying concern about the size of the TCB.

Split Up the TCB, Reduce the Privilege of Each Part: Murray *et al.* [116] removed Dom0 userspace from the TCB by moving the VM builder into a separate privileged VM. While a step in the right direction, it does not provide functional parity with Xen or remove the Dom0 kernel from the TCB, leaving the system vulnerable to attacks on exposed interfaces, such as network drivers.

Driver domains [51] allow device drivers to be hosted in dedicated VMs rather than Dom0, resulting in better driver isolation. Qubes-OS [132] uses

driver domains in a single-user environment, but does not otherwise break up Dom0. Stub domains [154] isolate the Qemu device model for improved performance and isolation. Xoar builds on these ideas and extends them to cover the entire control VM.

2.9 Discussion and Future Work

This idea of partitioning a TCB is hardly new, with software partitioning having been explored in a variety of contexts before. Microkernels remain largely in the domain of embedded devices with relatively small and focused development teams (e.g., [90]), and while attempts at application-level partitioning have demonstrated benefits in terms of securing sensitive data, they have also demonstrated challenges in implementation and concerns about maintenance [17, 24, 88, 128], primarily due to the mutability of application interfaces.

While fracturing the largely independent, shared services that run in the control VM above the hypervisor, we observe that these concerns do not apply to nearly the same degree; typically the components are drivers or application code exposing their dominant interfaces either to hardware or to dependent guests. Isolating such services into their own VMs was a surprisingly natural fit.

While it is tempting to attribute this to a general property of virtualization, we also think that it was particularly applicable to the architecture of Xen. Although implemented as a monolithic TCB, several of the components were designed to support further compartmentalization, with clear, narrow communication interfaces.

We believe the same is applicable to Hyper-V, which has a similar architecture to Xen. In contrast, KVM [89] converts the Linux kernel itself into a hypervisor, with the entire toolstack hosted in a Qemu process. Due to the tight coupling, we believe that disaggregating KVM this aggressively would be extremely hard, more akin to converting Linux into a microkernel.

2.9.1 Lessons

In the early design of the system our overall rule was to take a practical approach to hardening the hypervisor. As usual, with the hindsight of having built the system, some more specific guidelines are clear. We present them here as "lessons" and hope that they may be applied earlier in the design process of future systems.

Don't break functionality: From the outset, the work described in this chapter has been intended to be applied upstream to the open source Xen project. We believe that for VM security improvements to be deployed broadly, they must not sacrifice the set of functionality that has made these systems successful, and would not expect a warm reception for our work from the maintainers of the system if we were to propose that facilities such as CPU overcommit simply didn't make sense in our design.

This constraint places enormous limitations on what we are able to do in terms of hardening the system, but it also reduces the major argument against accepting new security enhancements.

Don't break maintainability: Just as the users of a virtualization platform will balk if enhancing security costs functionality, developers will push back on approaches to hardening a system that require additional effort from them. For this reason, our approach to hardening the hypervisor has been largely a *structural* one: individual service VMs already existed as independent applications in the monolithic control VM and so the large, initial portion of our work was simply to break each of these applications out into its own virtual machine. Source changes in this effort largely improved the existing source's readability and maintainability by removing hard-coded values and otherwise generalizing interfaces.

By initially breaking the existing components of the control VM out into their own virtual machines, we also made it much easier for new, alternate versions of these components to be written and maintained as drop-in replacements: our current implementation uses largely unchanged source for most of the service VM code, but then chooses to completely reimplement

XenStore. The original version of XenStore still works in Xoar, but the new one can be dropped in to strengthen a critical, trusted component of the system.

There isn't always a single best interface: The isolation of components into service VMs was achieved through multiple implementations: some service VMs use a complete Linux install, some a stripped-down "miniOS" UNIX-like environment, and some the even smaller "nanOS", effectively a library for building small single-purpose VMs designed to be amenable to static analysis.

Preserving application state across microreboots has a similar diversity of implementation: driver VMs take advantage of a recovery-box-like API, while for the reimplementation of XenStore it became more sensible to split the component into two VMs, effectively building our own long-lived recovery box component.

Our experience in building the system is that while we might have built simpler and more elegant versions of each of the individual components, we probably couldn't have used fewer of them without making the system more difficult to maintain.

2.9.2 Future Work

The mechanism of rebooting components that automatically renegotiate existing connections allow many parts of the virtualization platform to be upgraded in place. An old component can be shut down gracefully, and a new, upgraded one brought up in its place with a minor modification of XenStore keys. Unfortunately, these are not applicable to long-lived components with state like XenStore and the hypervisor itself. XenStore could potentially be restarted by persisting its state to disk. Restarting Xen under executing VMs, however, is more challenging. We would like to explore techniques like those in ReHype [95], but using *controlled* reboots to safely replace Xen, allowing the complete virtualization platform to be upgraded and restarted without disturbing the hosted VMs.

Although the overall design allows for it, our current implementation

does not include cross-host migration of VMs. We are in the process of implementing a new service VM that contains the live VM migration toolset to transmit VMs over the network. While this component is not currently complete, it has begun to demonstrate an additional benefit of disaggregation: the new implementation strikes a balance between the implementation of a feature that requires considerable privilege to map and monitor changes to a VM's memory in the control VM, and the proposal to completely internalize migration within the guest itself [33]. Xoar's live migration tool allows the guest to delegate access to map and monitor changes to its memory to a trusted VM, and allows that VM to run, much like the QemuVM, for as long as is necessary. We believe that this technique will further apply to other proposals for interposition-based services, such as memory sharing, compression, and virus scanning.

2.10 Conclusion

Advances in virtualization have spurred demand for highly-utilized, low-cost centralized hosting of systems in the cloud. The virtualization layer, while designed to be small and secure, has grown out of a need to support features desired by enterprises.

Xoar is an architectural change to the virtualization platform that looks at retrofitting microkernel-like isolation properties to the Xen hypervisor without sacrificing any existing functionality. It divides the control VM into a set of least-privilege service VMs, which not only makes any sharing dependencies between components explicit, but also allows microreboots to reduce the temporal attack surface of components in the system. We have achieved a significant reduction in the size of the TCB, and address a substantial percentage of the known classes of attacks against Xen, while maintaining feature parity and incurring very little performance overhead.

Chapter 3

Whose Cache Line Is It Anyway? Operating System Support for Live Detection and Repair of False Sharing

3.1 Introduction

Cache contention on modern CPUs can lead to performance collapse. This collapse is entirely workload dependent and cannot currently be mitigated by the hardware providing the caching. As a result, there exist applications that perform terribly on modern CPUs because they contend for cache lines. Not only are existing systems unable to correct this, they are also unaware of the very existence of such contention.

In modern CPUs, the design of processor caches is complicated by two properties. First, rather than increasing frequencies, processors are becoming more parallel. Second, cache coherence is still broadly held as a necessary property of CPU implementations. Increasing parallelism means that there are more threads operating on memory at once, while coherence demands that all threads see a single, consistent view of that memory. Where

concurrent accesses to the same cache line involve one or more writers, exclusive access is required and the resulting cache coherence protocol interactions necessitate expensive, synchronous notifications across multiple cores and even physical sockets in the system.

Avoiding cache line contention should be treated as a systems problem. The cache, after all, is a shared performance-critical resource and software layers such as the VMM, OS, and language runtime occupy a useful vantage point from which to mediate access. Unfortunately, identifying, understanding and resolving cache contention is a challenging task on modern CPUs. Once false sharing is identified, resolving it correctly requires a fine-grained remapping mechanism to "split" a cache line in a manner that allows concurrent threads to achieve non-contending access—a facility that is not provided by page-granularity MMU hardware.

The VMM-based prototype system described in this chapter achieves both of these goals. First, through a combination of hardware performance counters and memory virtualization, we present a *false sharing detection system* that is able to rapidly detect false sharing and identify the specific, relevant byte-level regions of data that are contending. Second, to resolve the contending accesses identified by our detector, we use both hardware page protection and binary instrumentation to introduce a *fine-grained memory remapper*. This facility extends conventional virtual memory support, which works at a page granularity, with a byte-level remapping facility. Using this interface, the system can elect to transparently move contending data structures in virtual memory into new locations in physical memory *while code actively executes* on the original virtual addresses.

We demonstrate that it is possible to detect and repair false sharing in a manner that works on existing hardware and applies to existing application binaries. Our detection system has sufficiently low overheads as to be deployed in both development and production environments, while the remapping engine transparently and efficiently redirects memory accesses, allowing data structures to be arbitrarily removed from the middle of a page and placed elsewhere in memory. The resulting system is capable of identifying and fixing false sharing in applications in under a second of execution,

resulting in a significant speedup for concurrent workloads.

3.2 Cache Coherence and Scalability

Cache coherent systems are parallel computing systems which, despite the presence of private, per-core caches, present a single, unified view of memory to the entire system at any given point in time. The benefits of such consistent, shared memory, especially in parallel programming, come at a scalability cost to the extent that several highly-parallel architectures [70, 151] and OSes [11] have explored system design in the absence of cache coherence. Still, many computer architects [106] and systems designers [21] believe that existing systems can, in fact, continue to scale to much greater degrees of parallelism.

Cache Coherence and the x86: Cache coherence on x86 processors is maintained using variants of the popular MESI state protocol [122]. In MESI, every cache line has a state associated with it, while the cache directory keeps track of the states and validity of different cache lines [65]. Simultaneous reads from multiple cores are supported by allowing multiple copies of the same cache line to coexist in "Shared" state in the individual private caches. Any write, however, causes the corresponding core to become the owner of a cache line, which is put in "Exclusive" or "Modified" state in its private cache, while all other copies are invalidated.

Subsequent requests from other cores are serviced by either the shared on-socket cache or by main memory. For local lines, i.e., those shared between cores on the same socket, the resulting flushes of modified data from private core caches are snooped for modified values, which are then written back before completing the request. Requests for remote lines are forwarded to the appropriate socket via the Quickpath Interconnect (QPI) [98].

Accesses to modified cache lines force a write-back to a location accessible to the requesting core: contention amongst on-socket cores updates the on-socket cache, while cross-socket cores are forced to write-back to main memory. As a result, latencies of accesses to contended memory vary signifi-

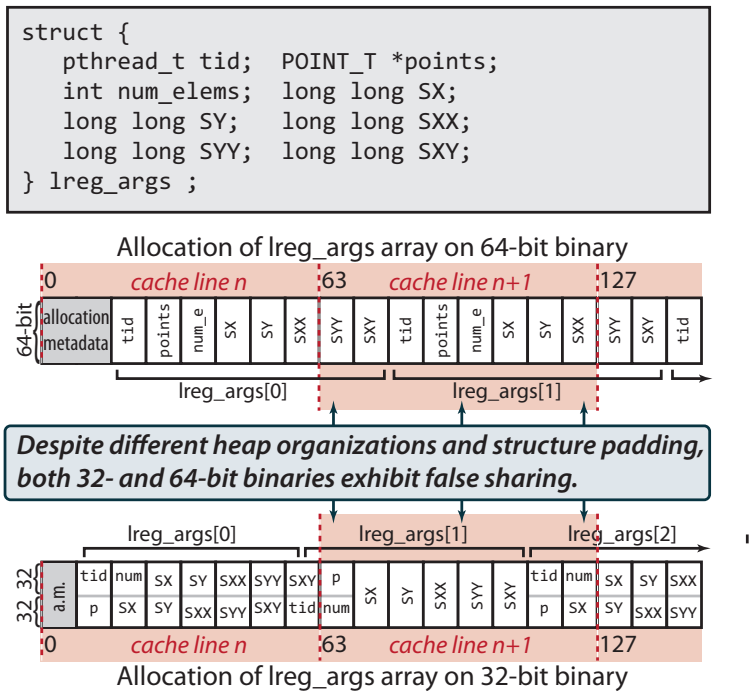

Figure 3.1: False sharing in the `linear_regression` benchmark

cantly depending on the exact physical topology of the cores involved [112].

True and False Sharing: Cache coherent architectures optimize for parallel workloads that tend to have large amounts of shared read-only data and smaller amounts of private mutable state. Cache lines with multiple accessors, at least one of which is a writer, experience expensive *coherence misses* as the coherence protocol must negotiate between cores in order to preserve consistency. True sharing occurs when concurrent accesses are to a single, shared data structure, such as a lock or reference count. False sharing occurs when independent data structures happen to reside on a single cache line; here, the workload matches the assumption of shared reads and isolated writes, but the granularity of isolation results in unnecessary contention.

The Phoenix [129] parallel benchmark suite's linear regression test is a popular example of false sharing [101, 166]. Figure 3.1 shows the `lreg_args`

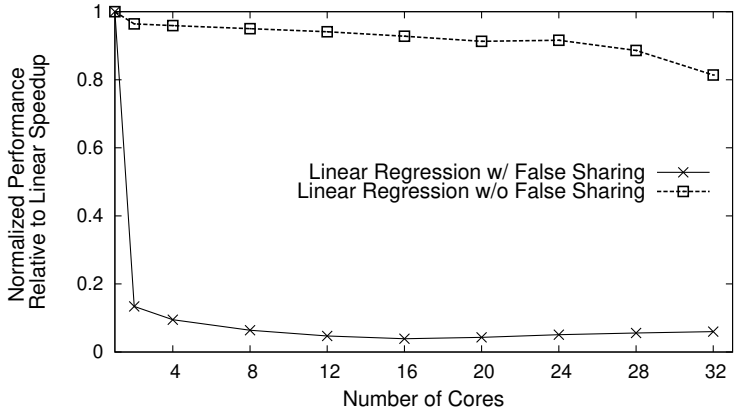

Figure 3.2: Effect of increased parallelism on performance

structure responsible for false sharing. An array of thread-indexed structures (`lreg_args[1..n]`) store mutable intermediate per-thread state (`SX, SY, SXX, SYY, SXY`) that is updated in a tight loop. This causes a high degree of false sharing when a thread's structure straddles cache lines and is co-located with another thread's structure. Figure 3.2 compares the program's scalability against a version modified to eliminate false sharing. While the modified version scales nearly linearly, adding additional cores to the original version often makes it *slower*, even in terms of absolute time. Another access pattern causing false sharing, seen in the Linux kernel [21], has a single frequently updated field in a structure surrounded by read-mostly data.

Besides the workload, false sharing depends on many dynamic properties in a system. Figure 3.1 shows the same source file compiled as both 32-bit and 64-bit binaries. Despite identical source and identical cache organization, the nature of false sharing is different: one case results in a 52-byte structure that tiles poorly across cache lines, whereas the other produces an ideally sized 64-byte structure, but then misaligns it because of allocator metadata.

False sharing is still a problem in today's systems: False sharing has long been recognized and studied as a problem on shared memory systems [20]. While compiler support can help in some cases, it is far from universal.[1] Many instances of contention are properties of workload and simply cannot be inferred statically. As evidence, recent years have seen significant examples of false sharing in mature, production software. False sharing has been seen in the Java garbage collector on a 256-way Niagara server [42], within the Linux kernel [21], and in spinlock pools within the popular Boost library [40, 107]. Transactional memory relying on cache line invalidations to abort transactions [66] also performs poorly with false sharing [113]. These examples serve as the basis for CCBench, the microbenchmark suite discussed in Section 3.6. That false sharing occurs in mature software is an indication not of a lack of quality, but rather that workloads leading to contention are often not seen in development and testing.

3.3 Design and Architecture

The system described in this chapter, called *Plastic*, provides system-level support to dynamically detect and mitigate persistent false sharing in unmodified application binaries. Plastic is a software implementation of a byte-granularity memory remapping mechanism. It allows any arbitrary byte range of virtual memory to be remapped from one physical location in memory to another, *while* the target is still running.

Specifically designed to mitigate false sharing, it determines the exact regions of virtual memory that currently exhibit contending accesses and then transparently remaps them to physical addresses on independent cache lines. Our approach is inspired by the sort of virtual-to-physical address remapping that is already possible with paging hardware, but refines it to a sufficiently fine granularity as to mitigate contention within a single cache line.

Modern x86 hardware does not support remapping at this fine granu-

[1] For example, `gcc` fixes false sharing in the Phoenix linear regression benchmark (see Figure 3.1) at `-O2` and `-O3` optimization, while `clang` fails to even at the highest optimization level.

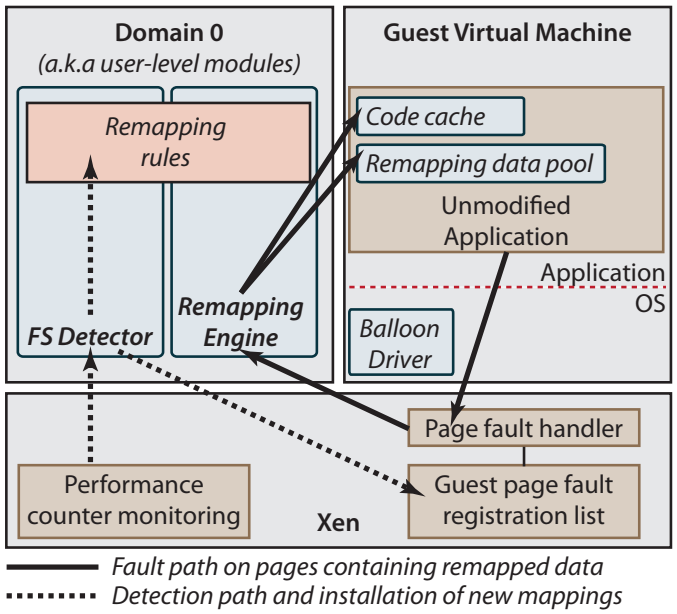

Figure 3.3: Plastic Architecture

larity. Plastic implements its own remapping system in software: provided with a set of byte granularity memory remapping rules, it applies them to a running system using live binary instrumentation. When false sharing is identified, Plastic creates a copy of the contending data on a non-contended cache line and uses dynamic instrumentation to redirect all accesses to that new location in memory.

Plastic is currently implemented on the Xen virtualization platform [8], where it takes advantage of memory interposition capabilities that are relatively easy to extend. It is important to emphasize that our approach is not specific to hypervisors: Plastic could be incorporated into an operating system with equal benefit. We use the term "operating system" in the title of this chapter to emphasize the more general opportunity for this class of system support.

Processors mask access latencies using instruction pipelining and out-of-order execution, thereby significantly reducing the impact of false sharing when contending accesses are separated by even a few hundred instruc-

tions. Plastic targets applications where performance is materially impacted by false sharing; typically, long running applications with high-frequency, parallel accesses to memory. To be practically deployable, it must function with low overhead, and detect and mitigate false sharing quickly and efficiently without requiring any changes to existing applications.

3.3.1 Architecture

Figure 3.3 shows Plastic's architecture at a high level. The bulk of the system resides in a user space tool running on a modified version of Xen. The hypervisor serves two purposes. First, the machine's hardware performance counters are exposed to monitor coherence invalidations on individual processor cores. Second, page protection interfaces are used to interpose on memory accesses and to determine the exact byte ranges involved in false sharing. Plastic's execution involves two main responsibilities: detection and remapping.

Detecting False Sharing: Detecting false sharing on x86 is a challenging task, especially given the constraint of imposing low overhead on the system. Plastic takes advantage of hardware performance counters to detect memory contention by monitoring coherence invalidation events, which indicate multiple cores competing for exclusive ownership of a cache line. Proceeding from this observation, it performs a series of progressively more expensive refinements, but applies them to increasingly specific regions of execution. This approach, described in detail in Section 3.4, allows Plastic to quickly detect false sharing and refine the diagnosis down to specific byte-granularity regions of contended memory.

Remapping Small Memory Ranges: On current x86 hardware, a page of memory—which is the smallest unit available to MMU hardware for remapping memory—contains 64 64-byte cache lines. Isolation requests from our false sharing detector are smaller regions *within* individual cache lines.

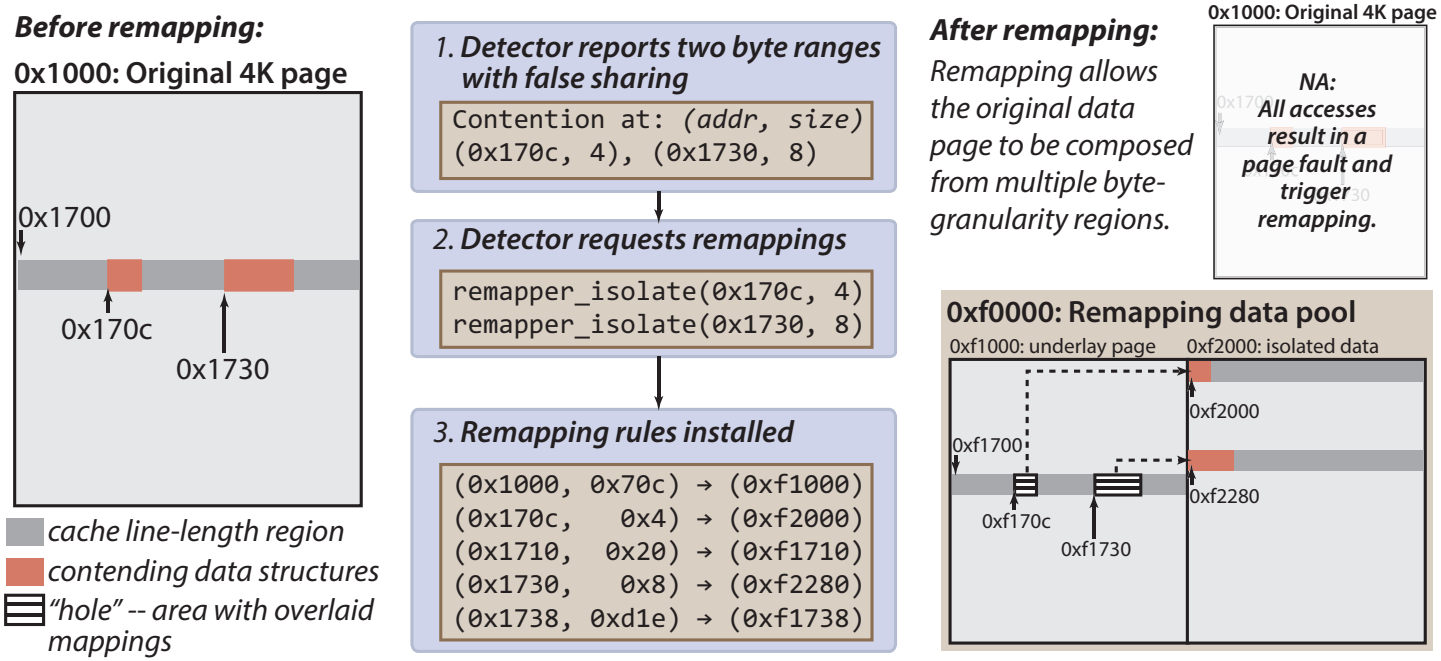

Figure 3.4: Byte-granularity remapping allows some data to be transparently isolated on separate cache lines.

Plastic achieves fine-grained remappings as illustrated in Figure 3.4. Isolation requests from the detector specify small regions of memory that should be placed on an isolated cache line. The remapper responds to isolation requests by providing sufficient remapping rules to describe the entire page containing remapped data. The result is that the page becomes a composite of one or more small isolated data ranges that are mapped on top of an underlay page. All of these data components reside in the remapping data pool, and the original page of virtual address space is marked as "No Access", resulting in faults on any attempt to read or write data through that range of virtual memory.

To make remapping efficient, demand faults on access to the original page result in the analysis and modification of accessing code to interact directly with the remapped data. Guard conditions in this modified code redirect only references that interact with remapped data, and leave other references interacting with original addresses in memory. By directly altering the accessing code, Plastic avoids expensive faults on future accesses while efficiently isolating the contending data onto independent cache lines.

Plastic's remapper takes advantage of a design property that is almost never available to dynamic instrumentation systems: *the implementation need not be complete with regard to instruction set semantics*. Plastic is free to modify code wherever capable of improving performance, however, as our goal is strictly to improve performance in the system, the option always exists to do nothing. In cases where Plastic is unable to safely or efficiently remap data ranges, it restores the data to the original page by copying back the composite regions, invalidates all active remappings for that data, and allows execution to continue against the original location. This places both the code and data back in the original and unmodified state. This observation, and the fact that page protection on the original data ensures that remapping covers all accesses to the remapped data from any code in the system, allow Plastic's design and implementation to pursue individual piece-wise optimizations based on workloads and access patterns that demonstrably benefit from remapping.

Plastic requires the guest OS to maintain both the code and data cache

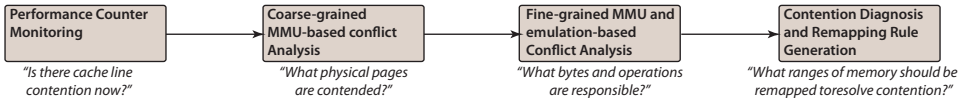

Figure 3.5: The stages of detecting and diagnosing cache contention

regions within the virtual address space of the application. This is the only requirement of Plastic on the guest and is achieved as an extension of the balloon driver, an in-guest memory management driver commonly installed in VMs. Under Windows, similar functionality can be obtained by the use of user-space AppInit DLLs.

3.4 Detection Methodology

Identifying false sharing typically requires a costly analysis of memory access patterns and to model those interactions within the cache hierarchy [60, 79, 166]. As shown in Figure 3.5, Plastic takes a multi-stage, sampling-based approach [25] to avoid these costs. Using a series of progressively more detailed (and consequentially higher-overhead) filters minimizes the impact of continuous detection and focuses the higher cost analysis on data likely to exhibit false sharing, based on information collected in earlier stages.

Starting from the left of the pipeline, we progressively refine the results, and begin by observing the presence of an abnormally large number of coherence invalidations using performance counters. We then isolate the pages where contention is occurring, before sampling memory accesses for short periods with emulation to find the precise regions of memory responsible for the contention.

3.4.1 Performance Counter Monitoring

Inputs: Running System
Outputs: Degree of contention in the system

Performance counters are special registers that store records of microarchitectural events, such as cache misses or branch prediction success rates, and are traditionally not used by the operating system and software. Acting

as free running counters, they can be used to investigate the performance characteristics of the system with low overhead. Modern processors have events to count messages at every level of the cache hierarchy, including the invalidation messages sent from one core to another due to contended accesses.

While coherence invalidations indicate the presence of false or true sharing, it is hard to characterize its impact on performance solely using an absolute count. This is because the performance impact of every invalidation is not equal: invalidations caused by off-socket cores require data to be fetched from main memory and are much more expensive than those from on-socket cores. However, since invalidations essentially stall a core until the data is fetched, contending on-socket cores can issue more requests, and hence cause more invalidations, per second than off-socket cores. Additionally, out-of-order execution and pipelining allow processors to hide the latency of such invalidations with useful execution.

Rather than using invalidation counts in isolation, Plastic quantifies their effect on performance by calculating the number of invalidations per instruction executed. As suggested in Intel's performance manuals, values over a third of a percent signify potentially high degrees of contention [73].

Invalidations are counted using the `SNOOP_RESPONSE_HITM` event, which counts the number of snoop requests to a particular core that "hit" a modified value in a private cache that now needs to be written back to maintain coherency. Plastic uses per-guest virtualized counters within Xen to only count invalidations that occur during guest execution. While cache line granularity contention does not distinguish between true and false sharing, an absence of such contention signifies the lack of significant false sharing in the system.

3.4.2 Page Granularity Analysis

Inputs: Presence of high cache line contention
Outputs: Contended physical pages

Once cache line level contention is observed, true and false sharing

are distinguished by determining the exact regions of memory accessed. Rather than instrumenting all memory accesses in the system, Plastic first determines contended pages by leveraging hardware page protection mechanisms.

As a hypervisor capable of running several unmodified guests simultaneously, Xen virtualizes memory and provides each guest the illusion of a contiguous address space. Hypervisor managed page tables perform an additional level of translation, either in software via shadow page tables or in hardware with Extended Page Tables (EPT) or Nested Page Tables (NPT) on Intel and AMD processors respectively.

Traditionally, Xen has a single set of hardware page tables per guest. Plastic extends these page tables to be per-core, each capable of having differing permissions for the same page. Using these tables to determine pages shared across cores is fairly trivial. Initially, all pages are set to "No Access" in all the private per-core page tables. Any subsequent access causes that core to fault, which promotes the permissions in its private page table and records the page and access-type in a per-core bitmap. Operating systems can perform similar analysis using per-thread page tables [14, 101].

Plastic periodically resets page permissions to determine the pages accessed by every core over several epochs. Since contended pages require at least one writer and one or more other readers or writers, the list of such pages for each sampling epoch is the intersection of the write bitmap of each core with the access bitmaps of all other cores. Contended pages, however, need not signify cache line contention since both thread migrations and non-overlapping heap objects on the same physical page could also be responsible.

3.4.3 Byte Level Access Analysis

Inputs: Contended physical pages

Outputs: Accessed bytes along with the identity of accessors

Analyzing memory accesses at a byte granularity requires recording every memory access. Plastic uses page table permissions to force a page fault

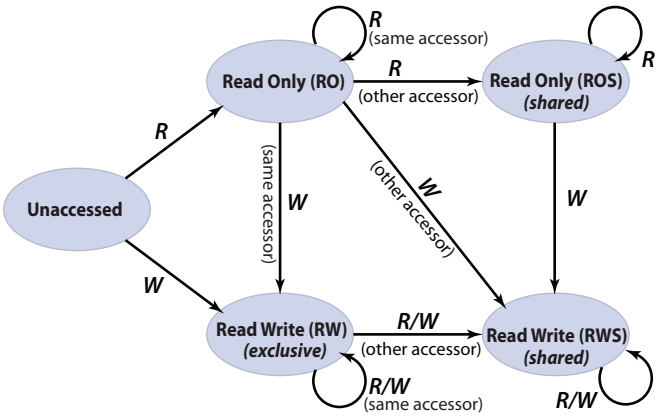

Figure 3.6: Sharing status of a byte in the access log

on any access to a contended page; the fault handler restores these permissions before returning to the guest, while simultaneously setting up a single-step breakpoint to force a trap as soon as the instruction is retired. At this point, permissions are again reset, effectively forcing *every* memory access on that page to fault, where it is logged for further analysis.

Access patterns, delineated on a per-core basis, are not necessarily good indicators of sharing since migrating threads may access the same regions of memory from different cores. Distinguishing accesses at a thread level requires some knowledge of how threading libraries identify different threads; for instance, since most threading use the `fs(x86_64)` or `gs(x86)` segment registers to select the descriptor for the Thread Local Storage (TLS), Plastic simply logs the descriptor value as a thread identifier.

3.4.4 Remapping Rule Generation

Inputs: Byte level access log for contended pages
Outputs: Remapping rules for the page

Generating remapping rules involves identifying contended cache lines by classifying individual bytes according to the number of accessors and the access type. Plastic parses the entire byte-level access log and assigns one of the states in Figure 3.6 to each byte. Contended cache lines have multiple

accessors with at least one writer (RW or RWS bytes).

During rule generation, memory regions within contended cache lines are grouped according to accessors. Multiple such groups are isolated from one another, while the bytes within the same group are remapped together. Groups with modified bytes (RW or RWS) are remapped to the isolated page, while the others are remapped to the underlay page.

3.4.5 Contention Verification and Adaptation

As a dynamic property, false sharing can evolve over time and requires constant monitoring and adaptation. Once the remapping rules are generated and sent to the remapping engine, the detection engine returns to monitoring performance counters for any contention in the system. Any detected contention triggers the entire detection pipeline which generates additional remapping rules as and when required.

Short lived false sharing may trigger detection and then subside before the actual remappings are applied. To avoid such scenarios, Plastic re-samples performance counters before remapping. While not guaranteeing that the previously detected false sharing exists, this ensures that some contention still exists before proceeding with the remappings.

3.5 Memory Remapping

Mitigating false sharing involves *transparently* and *safely* remapping *all* accesses to falsely shared regions of memory onto distinct cache line isolated locations. Plastic enforces such remappings with a combination of hardware-based page protection and dynamic binary instrumentation.

The remappings generated at the end of the detection pipeline only describe regions of contended memory without any mention of the corresponding accessors—in other words, they indicate what memory regions are to be remapped, but not where, in code, these remappings need to be applied. In order to detect all accessors, Plastic revokes access permission to contended pages and registers itself with the page fault handler for the entire lifetime of the remapping. Memory accesses are redirected using dynamic

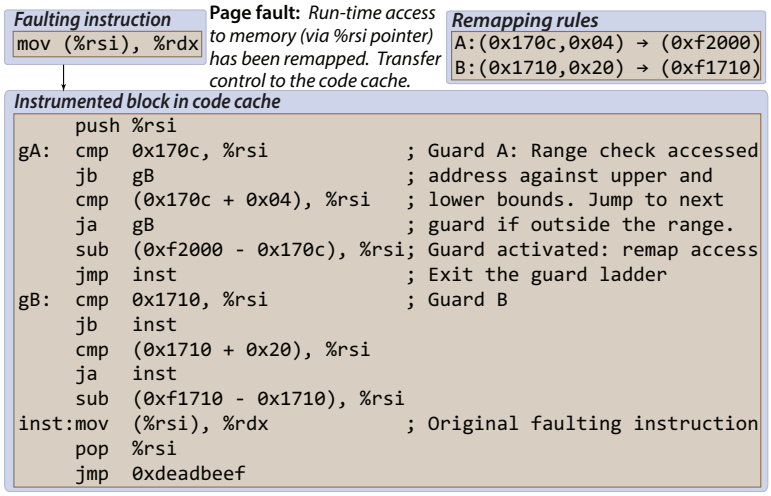

Figure 3.7: Guard Condition for a Single Instruction

instrumentation by "correcting" the accessing instruction.

Despite modifying the instructions executed, Plastic ensures that the semantics of the original code are preserved and the program remains safe at all times: instrumented instructions behave identically to the original with the exception of pointing to the remapped memory location.

3.5.1 Achieving Transparency

Plastic remaps arbitrary ranges of memory by redirecting memory accesses within a live, executing binary without any semantic knowledge of the program itself. While borrowing and adapting techniques from existing instrumentation and patching frameworks [15, 23, 103, 114, 119, 148], it does face two significant challenges. First, instrumenting a live binary cannot rely on any kind of load-time analysis; for example, several instrumentation frameworks [23, 103] generate a control flow graph and translate the entire program into basic blocks before executing it. Second, the kind of overheads acceptable in developer-facing diagnostic tools [114, 119] are not acceptable in performance-critical scenarios.

Plastic combines two techniques to apply the desired remappings: *fault*

driven redirection and *guard conditions*. Rather than rewriting the original instruction stream, it maintains a separate code cache with instrumented instructions. The instrumentation, applied with the help of DynamoRIO's [23] disassembly library, modify the memory referenced by the instructions while Plastic redirects execution flow in a way that oscillates between the original code and the code cache.

Fault Driven Redirection: Plastic maintains a consistent view of remapped memory for the entire system by redirecting every accessor to an instrumented version in the code cache. Statically identifying all accessors to a region of memory, however, is complicated, especially in the case of an already executing binary. Furthermore, redirecting execution using branches [148] is not possible on a variable instruction size architecture like x86 because adding a `jmp` or `call` as a trampoline could overwrite subsequent instructions and leave the code in an inconsistent state.

Plastic avoids these issues by leaving the original code unchanged and redirecting execution via faults on the data path. By revoking access permissions to contended pages, it forces any access to trigger a page fault and maintains a mapping between the faulting instruction and the instrumented code. Plastic then acts as a centralized dispatcher and redirects execution to the code cache by updating the instruction pointer.

Redirecting execution via the fault path also has another benefit: unlike code trampolines, instrumentation is restricted to instances of an instruction that access contended data, while other instances remain unchanged. This prevents cases where all callers of a library function suffer from instrumentation overhead, even when required only by a single caller.

Guard Conditions: When copied to the code cache, faulting memory references are replaced by code that includes, in addition to the original instruction, instrumentation to appropriately modify the address referenced by the instruction. Figure 3.7 illustrates an example of one such code block.

As such instructions may access different addresses depending on the context under which executed, updating the reference address to a fixed,

remapped location is insufficient. Instead, within the code cache, the instruction is preceded by a "guard condition", a set of checks similar to XFI [47].

Guard conditions verify that the memory referenced has been remapped and update the address according to the correct rule. References not matching existing rules fall through all the checks and execute the unmodified, original instruction, ensuring that program behaviour remains unchanged.

3.5.2 Optimizing Performance

Trampolining to the code cache via page faults is slow and routinely hitting the fault path for every contended memory access is several orders of magnitude slower than simply allowing false sharing to exist. Plastic attempts to reduce faults by instrumenting entire code blocks rather than single instructions, whenever possible.

Almost all high-frequency, performance-impacting contended accesses, including calls and inlined accessors, are wrapped by loops. Instrumenting entire loops within the code cache amortizes the fault cost over several iterations. The primary exception to this case, i.e. high-frequency contended accesses in straight line code, is code that is called repeatedly through asynchronous event injection: syscalls, interrupts, and user-level equivalents such as signal handlers. Plastic can be extended to optimize this case by instrumenting the entire function and then rewriting function call sites [119].

Code blocks are instrumented by guarding memory references to ensure that they are correctly remapped. These code blocks terminate with a direct `jmp` back to the next instruction in the original code. Branch target offsets and `rip`-relative accesses are corrected to account for both the code relocation and the added instrumentation instructions.

Identifying Code Blocks: To instrument an entire code block, rather than an individual instruction, Plastic faces the challenge of identifying block boundaries based only on an instruction pointer lying somewhere within the block. Fortunately, modern processors have the ability to track both

source and destination addresses for recently taken branches using a facility called the Last Branch Record (LBR). Plastic identifies blocks by searching this LBR at the time of a fault for recent branches that move backwards in code and describe an address range containing the faulting instruction.

Despite the simplicity of this approach, it has proven to be very effective in practice: even with a typically 16-entry LBR history, Plastic is able to identify loop boundaries that effectively allows it to amortize the overhead of transferring execution to the code cache. Faulting instructions within hot code blocks cause the entire block to be instrumented while other accesses are instrumented on a per-instruction basis, preserving remappings for less-frequently accessors. In the future, we anticipate extending Plastic to periodically sample the LBR in the background to better detect nested loops and loop-embedded function calls.

Specialized Code Blocks: As implied by Figure 3.7, memory references within the code cache require a guard condition for *every* applied remapping. This approach, especially in the face of large numbers of mappings, is problematic: threads typically access only a small subset of the remapped data, similar to the example of adjacent private structures in Figure 3.1, and forcing execution through a long ladder of conditional guards increases the general-case overhead of remapping. Even worse, a small number of "straggler" threads may end up falling through *all* the guard conditions; the corresponding increase in overhead delays the entire program and loses the performance benefit for all other threads.

Plastic takes advantage of the locality of accesses within threads by *not* instrumenting blocks to contain a comprehensive list of guard conditions. Instead, individual thread-specific versions of a code block are generated to contain exactly the guards necessary to handle the remapping of data accessed by that specific thread. As a result, threads executing their specialized version evaluate as few conditionals as possible and fault on accesses when an appropriate guard is missing. This fault may then result in the generation of an alternate block containing the necessary additional guards.

Transferring Execution: While transferring execution from the original code to the code cache is trivial in the case of a single instruction, entire code blocks pose two problems to execution transfer. First, rather than transferring execution to the start of the instrumented block, it has to be transferred to the correct instruction within it. Second, for multi-threaded applications, Plastic must safely migrate all threads concurrently executing within that code block.

Plastic maintains mappings between the original code and the code cache using the Translation Offset Index. During page faults, it selects a specialized block with guards for the faulting address and transfers execution to the instruction corresponding to the faulting instruction in that block. The offset index allows faults on *any* instruction in the original binary to be appropriately redirected to the code cache. This is important because at the time a remapping is applied, several worker threads may be executing different instructions within the code block being remapped.

Figure 3.8 illustrates this transfer for two threads, T_1 and T_2 that contend on independent, thread-specific data. T_1 accesses data described by rule A and triggers a page fault that generates a specialized version of the code block with guards for both rules A and C. The second guard is added because it describes the common case of accesses to the remainder of the original page. Meanwhile, T_2 continues executing the original code until it faults on a memory reference, in this case one associated with rule B; Plastic then generates a new specialized block for rules B and C, and transfers execution to it. In this regard, the combination of faults on data access and offset index matching between the original and specialized versions of code allow threads to be efficiently and safely redirected to the suitable specialized version.

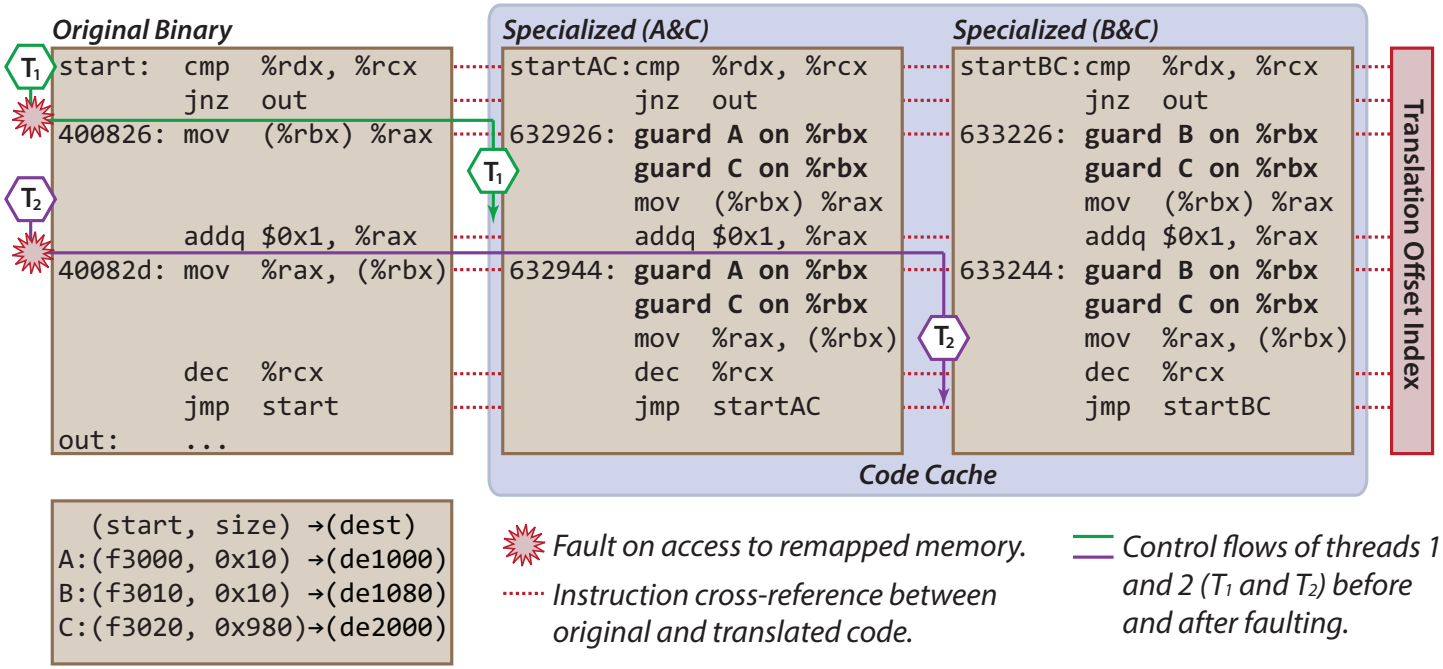

Figure 3.8: Control transfer on access fault from original binary, to specialized blocks in the code cache

3.5.3 Safety of Instrumented Code

Plastic guarantees that applied code transformations do not alter program functionality in any manner. To ensure this, it performs extremely simple transformations: every faulting instruction in the original code has an identical counterpart in the code cache. Instrumentations simply modify the referenced addresses and invariants that hold for the original execution also hold for the version within the code cache.

Plastic leverages the insight that, as a performance optimization, it can afford to be sound, but not complete, with respect to the instruction set. While systems providing strong security guarantees [49, 165] have to contend with several corner cases in x86, Plastic simply invalidates all the remappings when it encounters instructions it cannot safely redirect.

Invalidating remappings is simplified by the fact that the original code remains unchanged at all times. Data from the split pages is merged back to the original which is then unprotected. Execution transfer is mirrored from earlier—permission to the split pages is revoked and Plastic transfers accessors from the code cache back to the corresponding accessors in the original code. With both code and data restored to their original state, execution continues unhindered.

Code Coverage: As discussed in Section 3.5.1, Plastic prevents stray accessors from modifying the original data by marking the page as "No Access" throughout the lifetime of the remapping. Guard conditions ensure that accesses to data lying outside the remapped region, even from within the code cache, do not get arbitrarily remapped.

Thread Safety: Plastic serializes handling page faults for the data page and the subsequent generation of instrumented code blocks. Such faults are infrequent, so serialization does not significantly affect performance, while simultaneously preventing race conditions due to concurrent accesses.

Leaked Pointers: Applications remain unaware of the remapped data ranges and as such all accesses to these ranges originate from within the code

cache. Improperly restoring register state, however, may reveal these ranges to the original code. Such leaked pointers are dangerous: an application may inadvertently manipulate them and attempt to access undefined regions of memory. Plastic avoids leaked pointers by immediately restoring the original memory address after executing the faulting instruction.

Unsupported Instructions: Plastic invalidates existing remappings on encountering memory references that it cannot redirect. A list of several such instructions and the difficulties involved in their redirection follows. While workarounds for several of these instructions exist, they are currently not implemented in Plastic.

1. Repeat Prefixed Instructions: Instructions like `rep movs` and `rep cmps` may access several bytes of memory, spanning both remapped and non-remapped regions, using a single instruction. Plastic cannot detect the ranges of memory involved and redirect accesses just by modifying the parameters prior to execution. Instrumentation frameworks like Pin [103] explicitly convert such instructions into loops to overcome this issue.

2. Atomic Accesses: Memory accesses that straddle remap regions cannot safely be redirected without changing the instruction into a set of smaller granularity memory accesses. Examples of such scenarios include contiguous bytes updated by independent threads, causing them to be remapped to independent cache lines, that are later read by a single atomic 32 or 64-bit read.

3. `mov %rXx, (%rXx)`: Directly modifying instructions that write their address to their memory location lead to leaked pointers as remapped addresses get written back to memory. Such instructions can be redirected by using an extra register to hold the original address which is written to the remapped memory region.

4. Register Indirect Branches: Indirect branches, such as `jmp *%rXx` and `call *%rXx`, accessing function pointers tables located on pro-

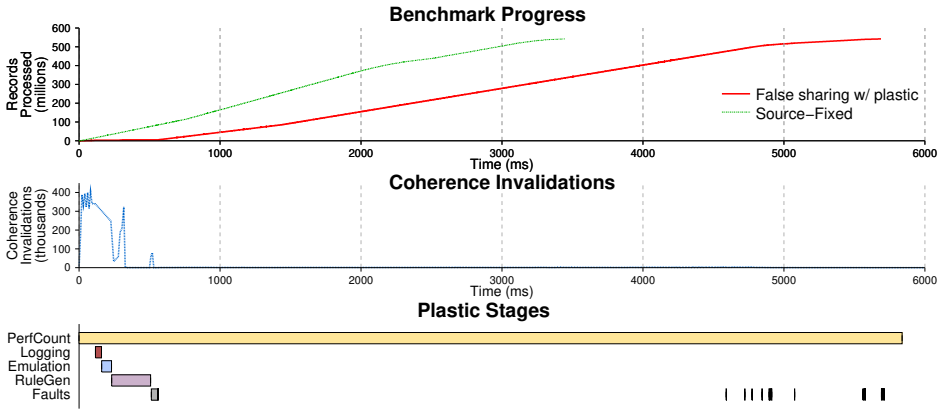

Figure 3.9: Linear regression running under Plastic

tected pages need to be redirected to the remapped pointer table. Once the branch is taken, however, Plastic cannot restore the register to its original value. Such branches can be replaced with `rip`-relative branches referencing the remapped pointer table.

Lastly, there is the safety of the code cache itself to consider. The code cache must reside in the address space of the application and remain accessible at all times; its pages, however, are marked as read-only to prevent modification from within the application. While an application may attempt to jump into the middle of a block to avoid the guards, there is little value in this as it would only hinder the application itself.

3.6 Evaluation

Plastic is evaluated on a dual socket, 8-core Nehalem system with 32 GB of memory. Each processor is a 4-core 64-bit Intel Xeon E5506 with private, per-core L1 and L2 caches and a shared, per-socket L3 cache. Plastic runs on Xen 4.2 with a Linux Dom0. All tests are run on an 8-core guest with virtual processors pinned to the corresponding physical core.

First, we describe the detailed execution of a false sharing workload under Plastic, followed by a discussion of the memory overhead of the system.

We then assess the performance impact of remappings within a code block on other callers of the same code block. Finally, we evaluate Plastic's performance across a range of different workloads, compiled with `gcc 4.4.3` at the default optimization level. Reported performance results are the average of twenty runs.

3.6.1 Functioning of Plastic

Detection Times and Execution under Plastic: Figure 3.9 represents execution of the `linear_regression` workload discussed earlier running under Plastic and compares it with that of a source-fixed version of the same program. Along with total benchmark progress, coherence invalidations are also shown for the version with false sharing.

As the workload starts it immediately causes a significant number of coherence invalidations; correspondingly, its throughput is only a fraction of that of the source-fixed version. At around 125 ms, the performance counters detect the presence of contention and activate the rest of the pipeline. At around 500 ms, the remapping rules are synthesized and threads are migrated to the code cache by 600 ms through a series of page faults. Execution remains within the code cache, as indicated by the absence of any further faults. Consequently, throughput rises and the benchmark progresses rapidly, while the coherence invalidations correspondingly drop to almost zero indicating the lack of contention.

A single thread aggregating results from remapped data is responsible for the page faults near the end of the execution. As these are non-high-frequency accesses, Plastic remaps them on a per-instruction basis, resulting in the high number of faults. Plastic continues to sample performance counters throughout execution for any further instances of contention; in this case, however, no other contention is detected.

Comparing the throughput of the Plastic-fixed and source-fixed versions helps precisely define the overhead of the extra instrumentation required for remapping. Once execution is transferred to the code cache, the throughput of the Plastic-fixed version is 110M/s compared to a throughput of 160M/s

for the source-fixed version—a performance loss of 31%. The remaining difference in overall throughput is due to the false sharing before the remappings are applied.

Memory Overhead: Plastic trades memory for higher performance by requiring additional memory for the instrumented code and to pad and isolate contended cache lines. In practice we find that a reservation of 64 pages (256 KB) is sufficient for most remapping scenarios and does not significantly reduce the VA space available to applications.

Impact of Remappings on Other Callers: A simple microbenchmark helps verify the assertion in Section 3.5.1 that callers of code blocks not referencing remapped data are unaffected: a function executes referencing non-contended memory followed by an execution referencing contended memory that triggers remapping. Finally, the original execution is repeated. Non-coherence misses are ignored since data resides on a single cache line. The last execution has under 1% overhead compared to the first, demonstrating the negligible impact on callers not accessing remapped data.

3.6.2 Performance Analysis

Plastic is evaluated by comparing the performance of several workloads running under Xen, normalized against their single threaded performance, with and without Plastic. While this data, shown in Figure 3.10, ignores the virtualization overhead, we find it to average 3% over the benchmarks.

Plastic samples contended pages for multiple 2 ms epochs, followed by 250 ms of emulation. Stage lengths can be varied if desired and, like in any sampling, represent a trade-off between the speed and accuracy of detection. Nevertheless, we believe they are sufficient for most high-frequency false sharing and are, in practice, able to accurately detect the exact regions of false sharing in all the workloads evaluated.

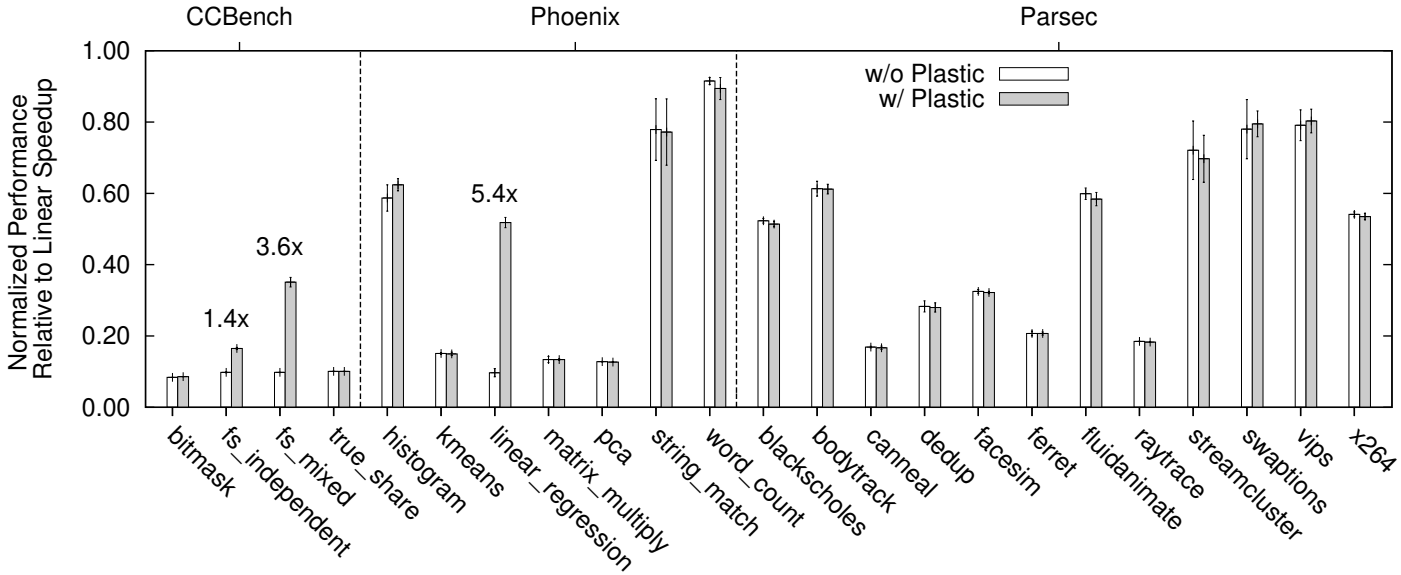

Figure 3.10: Performance of Phoenix, Parsec and CCBench suites running with Plastic.

Name	Description	Examples	Fixable?
`fs_independent`	Multiple accessors to independent variables (At least one writer)	Linear Regression in Phoenix [129] `spinlock_pool` in Boost [107] Bookeeping in the Java GC [42]	Yes
`fs_mixed`	Shared read-only data co-located with contended data	`net_device` struct in Linux [21]	Yes
`bitmask`	Bitmasks and flags	`page` struct in Linux [21]	No
`true_share`	Shared read-write data	Locks and global counters	No

Table 3.1: Microbenchmarks in CCBench. The Fixable column denotes whether it can be fixed by simply remapping memory.

The CCBench: Performance artifacts due to memory contention in real workloads, like those in Section 3.2, often only manifest themselves at a not-yet-common degree of scale. To replicate these effects with fewer cores, we have developed *CCBench*, a suite of microbenchmarks that model their memory access patterns on real workloads, but exacerbate their effects by contending at higher frequencies.

Table 3.1 briefly describes the different microbenchmarks and the real workloads they emulate, while their performance is seen in Figure 3.10. Including both true and false sharing workloads allows us to measure both the performance improvement when Plastic is able to "fix" the problem as well as the impact on performance when it is unable to do so.

`fs_independent` is a classical example of false sharing with multiple readers and writers accessing independent values in a global array. It is modeled after spinlocks in a lock pool [107] or the bytes used to represent the dirtied status of pages during Java garbage collection [42]. Isolating the sets of values accessed by a thread onto independent cache lines reduces execution time from 18.4s to 5.1s, a speedup of 3.6x.

`fs_mixed` involves false sharing between a shared read-only data range and a shared read-write range, with accessors equally distributed between both data ranges. Read-mostly spinlocks in the `net_device` structure in Linux contend with the transmission and receive queues in a similar manner [21]. Remapping splits these data ranges and reduces execution time from 18.7s to 11s, a modest performance improvement of 40%. Comparing the performance of accessors now shows a bimodal distribution: accessors to the read-only range are contention-free and display a speedup of 6.6x, but overall program performance is limited by accessors to the read-write range still suffering from some contention.

`bitmask` models bitmasks like the flags in the `page` structure in the Linux kernel. A combination of read-only and read-write flags leads to false sharing *within* a single byte, alleviated only by splitting the flags into discrete versions [21]. From a byte-level perspective, however, bitmasks represent true sharing and do not benefit from remapping.

`true_share` simulates lock contention with concurrent reads and writes

to the same memory location.

`bitmask` and `true_share` represent pathological scenarios for Plastic: high degrees of true sharing that are impossible to distinguish from false sharing without emulation. Despite this, due to the sampled nature of emulation, both show little overhead and perform within 1% of normal execution.

Shared Memory Benchmarks: Plastic's effectiveness with real-world benchmarks is evaluated against several applications from the Phoenix [129] and PARSEC [16] benchmark suites. Both of these suites are specifically designed for shared memory workloads and are representative of applications from several different domains.

Plastic fixes significant amounts of false sharing in `linear_regression`, showing a speedup of 5.4x compared to normal execution. For the remaining workloads, Plastic imposes an average of 3% overhead, demonstrating that it does not adversely impact workloads without false sharing.

As a live detector, Plastic prioritizes performance over completeness and focuses on detecting high-impact false sharing. While this does lead to some instances of false sharing remaining undetected, we quantify the degree of false sharing in these instances by comparing against other detectors. Sheriff [101] detects false sharing in `streamcluster`, `swaptions`, `histogram`, `string_match`, and `word_count` in addition to `linear_regression`.

Out of these, `streamcluster`, `swaptions`, and `word_count` show less than 5% degradation due to false sharing and do not have enough contention to trigger the detection pipeline. False sharing in `string_match` is caused due to the heap allocator. Not only is this not observed on our system, but in practice it scales well up to 8 cores.

In contrast, while `histogram` definitely suffers from false sharing, the performance impact is found to be only around 25% when compared against a source-fixed version. This does not warrant remapping because, unlike the case in `linear_regression`, `fs_independent`, and `fs_mixed`, the benefit of mitigating false sharing no longer masks the remapping overhead

discussed in Section 3.6.1.

The benchmark results highlight the differences in approach taken to false sharing detection and mitigation by Sheriff [101] and Plastic. Sheriff forces threads within a process to operate on private pages, set up using copy-on-write semantics, and merges them at synchronization points. False sharing is avoided simply by batching updates to contended memory regions. `linear_regression` has little synchronization and exhibits a 9x speedup. In contrast, Plastic has a low throughput detection phase prior to applying the remappings and shows a speedup of 5.4x. By avoiding expensive page copy operations, Plastic does an excellent job of uniformly imposing low overhead on workloads that do not exhibit false sharing. In contrast, Sheriff shows significant overheads in programs with frequent locking such as `fluidanimate` and `canneal`.

We find that Plastic can quickly and accurately detect and correct false sharing with low overhead. In cases where false sharing exists, but imposes only a small overhead, it is able to correctly value its potential to improve performance and do no harm. At the same time, Plastic can significantly improve the execution of workloads where false sharing would otherwise impose a crippling scalability limitation.

3.7 Related Work

Several existing systems study the cache subsystem and detect false sharing in existing application workloads. Plastic also shares similarities in detection mechanisms with race detectors and other systems designed to diagnose memory contention issues apart from false sharing.

False Sharing: Sheriff [101] shares a similar goal to Plastic in transparently detecting and fixing false sharing in production environments. It splits threads into separate, independent processes, each of which has private page tables. Changes to memory are localized to a private copy of modified pages which are merged together at synchronization points. False sharing is detected by identifying interleaved writes at a cache line granular-

ity, while mitigation simply reduces the frequency of accesses to contended cache lines.

Sheriff makes assumptions about the use of the `pthread` API for synchronization and may break correctness if these assumptions are violated; for instance, in the case of lock-free data structures. Similarly, locks and other synchronization primitives located on the same page as contended structures may prevent the successful mitigation of false sharing.

Zhao, *et al.* [166] use memory shadowing to track ownership of cache lines and analyze thread-access patterns without full cache simulation. Using DynamoRIO [23] for instrumentation, they help detect cache contention with around 5x overhead, and makes no attempt to mitigate the problem.

DProf [124] is a data profiler that associates access costs with data rather than instructions. It helps developers identify memory regions frequently experiencing high access costs, including due to true and false sharing, but does not automatically distinguish between the causes or help mitigate them.

Intel Performance Tuning Utility [75] uses performance counter monitoring to identify contention, but does not distinguish between true and false sharing. Unlike Plastic, it neither attempts to analyze the performance effects of this contention, nor does it attempt to fix them in any way.

Several approaches for detecting cache contention model the cache in software [60, 79, 150], and are clearly intended for development use only. Pluto [60], a Valgrind based instrumentation engine, imposes around two orders of magnitude overhead, while Cmp$im [79], which uses Pin [103] to simulate the entire memory hierarchy and coherence algorithms in software, runs at 4-10 MIPS.

Race Detection: Like detecting false sharing, race detection requires instrumenting every memory access to log the ordering of accesses to shared memory. Aikido [120] uses hypervisor-based, per-thread shadow page tables to identify contended pages for further analysis, in a manner similar to Plastic. Greathouse, *et al.* [55] use coherence invalidations as a trigger for more heavy-weight, software-based analysis to identify actual data races. As a

race detector, however, they treat any such invalidation as suspicious and perform further analysis, without analyzing its performance impact.

3.8 Discussion and Future Work

In order to maximize performance, Plastic leverages hardware facilities whenever possible, only resorting to software interposition when the required features are not exposed by the hardware. Processors, however, are constantly evolving and some additional features may further reduce overhead.

Performance counters operating in sampling mode, called Precise Event Based Sampling (PEBS) or Instruction Based Sampling (IBS) on Intel and AMD processors respectively, store processor state on sampled occurrences of selected microarchitectural events. Sampling coherence invalidations stores processor state when contended memory accesses occur. Unfortunately, on Nehalem processors this does *not* include the memory address accessed. Recording this address, as proposed for future architectures, would allow Plastic to identify contended memory regions directly in hardware.

Plastic relies on page-granularity protection to detect access to contended memory, unfortunately resulting in faults for all accesses to that page. Hardware watchpoints overcome this by detecting accesses at byte-granularity, but are extremely limited in number. An unlimited number of watchpoints [56] would help significantly reduce the number of faults.

Language runtimes that manage memory for their applications independently of the OS could suffer unnecessary performance degradation due to remapping. Plastic invalidates applied remappings to maintain program safety, but does not currently extend this facility to monitor the performance impact of the remapping and, if necessary, restore execution to the original code. Plastic also cannot be disabled on a per-application basis, but could be extended to provide this facility with support from the guest OS.

Lastly, dynamically fixing false sharing should be a last resort only for when statically fixing the problem in source is not possible. Plastic's detection engine could be extended with debug symbols to provide developers source-level reports of the contending data structures.

3.9 Conclusion

Plastic demonstrates that, by taking responsibility for monitoring and managing coherence misses in its caches, a system can dynamically recover from workloads that exhibit pathological false sharing. In order to achieve this, the system tackled two challenging problems. First, the aggregation and integration of a variety of monitoring and diagnosis techniques, including hardware performance counters, shadow paging, and instruction emulation to quickly and precisely identify false sharing with low overhead. Second, the system demonstrated a sub-page granularity remapping facility that is sufficiently high-performance as to show a speedup of 3–6x in cases of high-rate false sharing.

Chapter 4

Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage

4.1 Introduction

Rack-scale storage architectures such as Facebook's Lightning [126] and EMC's DSSD [46] are dense enclosures containing storage class memories (SCMs)[1] that occupy only a few units of rack space and are capable of serving millions of requests per second across petabytes of persistent data. These architectures introduce a tension between efficiency and performance: the bursty access patterns of applications necessitate that storage devices be shared across multiple tenants in order to achieve efficient utilization [91], but the microsecond-granularity access latencies of SCMs render them highly sensitive to software overheads along the datapath [30, 147].

This tension has forced a reconsideration of storage abstractions and raised questions about where functionality, such as virtualization, isolation, and redundancy, should be provided. How should these abstractions

[1]We use the term storage class memory through the rest of the chapter to characterize high performance PCIe-based NVMe SSDs and NVDIMM-based persistent storage devices.

evolve to support datacenter tenants without compromising efficiency, performance, or overall system complexity?

Decibel is a thin virtualization platform, analogous to a processor hypervisor, designed for rack-scale storage, that demonstrates the ability to provide tenants with flexible, low-latency access to SCMs. Its design is motivated by the following three observations:

1. Device-side request processing simplifies storage implementations. By centralizing the control logic necessary for multi-tenancy at processors adjacent to storage devices, traditional storage systems impose latency overheads of hundreds of microseconds to few milliseconds under load [91]. In an attempt to preserve device performance, there has been a strong trend towards bypassing the CPU altogether and using hardware-level device passthrough and proprietary interconnects to present SCMs as *serverless storage* [5, 30, 68].

Serverless storage systems throw the proverbial baby out with the bathwater: eliminating the device-side CPU from the datapath also eliminates an important mediation point for client accesses and shifts the burden of providing datapath functionality to client-based implementations [7, 29] or to the devices themselves [2, 68, 163]. For isolation, in particular, client implementations result in complicated distributed logic for co-ordinating accesses [68] and thorny questions about trust.

2. Recent datacenter infrastructure applications have encompassed functionality present in existing feature-rich storage abstractions. In addition to virtualizing storage, storage volumes provide a rich set of functionality such as data striping, replication, and failure resilience [26, 50, 71, 123]. Today, scalable, cloud-focused *data stores* that provide persistent interfaces, such as key-value stores, databases, and pub/sub systems, increasingly provide this functionality as part of the application; consequently, the provision of these features within the storage system represents a duplication of function and risks introducing both waste and unnecessary overheads.

3. Virtualizing only the capacity of devices is insufficient for isolation in multi-tenant environments. Extracting performance from SCMs is extremely compute-intensive [30, 121, 164] and sensitive to cross-core con-

tention [18]. As a result, storage systems require a system-wide approach to virtualization and must ensure both adequate availability of compute, network, and storage resources for tenant requests, and the ability to service these requests in a contention-free manner. Further, unlike traditional storage volumes that do not adequately insulate tenants from performance interference [91], the system must provide tenants with predictable performance in the face of multi-tenancy.

These observations guide us to a minimal storage abstraction that targets isolation and efficient resource sharing for disaggregated storage hardware. Decibel introduces *Decibel volumes* (referred to as *dVols* for short): vertical slices of the storage host that bind SCMs with the compute and network resources necessary to service tenant requests. As both the presentation of fine-grained storage abstractions, such as files, objects, and key-value pairs, and datapath functionality, such as redundancy and fault-tolerance, have moved up the stack, dVols provide a minimal consumable abstraction for shared storage without sacrificing operational facilities such as transparent data migration.

To ensure microsecond-level access latencies, Decibel prototypes a runtime that actively manages hardware resources and controls request scheduling. The runtime partitions hardware resources across cores and treats dVols as schedulable entities, similar to threads, to be executed where adequate resources are available to service requests. Even on a single core, kernel scheduling policies may cause interference, so Decibel completely bypasses the kernel for both network and storage requests, and co-operatively schedules request processing logic and device I/O on a single thread.

Decibel is evaluated using a commodity Xeon server with four directly-connected enterprise PCIe NVMe drives in a single 1U chassis. Decibel presents storage to remote tenants over Ethernet-based networking using a pair of 40 GbE NICs and achieves device-saturated throughputs with a latency of 220–450 μs, an overhead of approximately 20–30 μs relative to local access times.

Application	Backend	FT	Ephem	Year
Key-Value Stores				
Riak	LevelDB	Y	Y	2009
Voldemort	BerkeleyDB	Y	N	2009
Hyperdex	File	Y	Y	2011
Databases				
Cassandra	File	Y	Y	2008
MongoDB	WiredTiger	Y	N	2009
CockroachDB	RocksDB	Y	Y	2014
Pub/Sub Systems				
Kafka	File	Y	Y	2011
Pulsar	BookKeeper	Y	Y	2016

Table 4.1: Examples of cloud data stores. *FT* denotes that replication and fault-tolerance are handled within the data store and storage failures are treated as node failures. *Ephem* indicates that users are encouraged to install data stores over local, "ephemeral" disks—Voldemort and Mongo suggest using host-level RAID as a convenience for recovering from drive failures. Backend is the underlying storage interface; all of these systems assume a local filesystem such as ext4, but several use a library-based storage abstraction over the file system API.

4.2 Decibel and dVols

Scalable data stores designed specifically for the cloud are important infrastructure applications within the datacenter. Table 4.1 lists some popular data stores, each of which treats VM or container-based nodes as atomic failure domains and handles lower-level network, storage, and application failures uniformly at the node level. As a result, several of these data stores recommend deploying on "ephemeral", locally-attached disks in lieu of reliable, replicated storage volumes [37, 93, 109].

These systems are designed to use simple local disks because duplicating functionality such as data redundancy at the application and storage layers is wasteful in terms of both cost and performance; for example, running a data store with three-way replication on top of three-way replicated storage results in a 9x write amplification for every client write. Further, running a replication protocol at multiple layers bounds the latency of write requests

to the latency of the slowest device in the replica set. The desire to minimize this latency has led to the development of persistent key-value stores, such as LevelDB and RocksDB, that provide a simple, block-like storage abstraction and focus entirely on providing high performance access to SCMs.

The dVol is an abstraction designed specifically for rack-scale storage in response to these observations. As a multi-tenant system, Decibel faces challenges similar to a virtual machine monitor in isolating and sharing an extremely fast device across multiple tenants and benefits from a similar lightweight, performance-focused approach to multiplexing hardware. Correspondingly, a dVol is a schedulable software abstraction that encapsulates the multiple hardware resources required to *serve* stored data. In taking an end-to-end view of resource sharing and isolation rather than focusing only on virtualizing storage capacity, dVols resemble virtual machines to a greater degree than traditional storage volumes.

In borrowing from VMs as a successful abstraction for datacenter computation, dVols provide the following properties for storage resources:

Extensible Hardware-like Interfaces: dVols present tenants with an interface closely resembling a physical device and so avoid restricting application semantics. dVols also offer tenants the ability to offload functionality not directly supported by SCMs. For example, dVols support atomic updates [36, 67] and compare-and-swap [156] operations.

Support for Operational Tasks: Decoupling storage from the underlying hardware provides a valuable point of indirection in support of datacenter resource management. In virtualizing physical storage, dVols support operational tasks such as non-disruptive migration and provide a primitive for dynamic resource schedulers to optimize the placement of dVols.

Visibility of Failure Domains: Unlike traditional volumes that abstract information about the underlying hardware away, dVols retain and present enough device information to allow applications to reason about failure domains and to appropriately manage placement across hosts and devices.

Elastic Capacity: SCMs are arbitrarily partitioned into independent, non-contiguous dVols at runtime and, subject to device capacity constraints, can grow and shrink during execution without causing device fragmentation

and a corresponding wastage of space.

Strong Isolation and Access Control: As multi-tenant storage runs the risk of performance interference due to co-location, dVols allow tenants to specify service-level objectives (SLOs) and provide cross-tenant isolation and the throughput guarantees necessary for data stores. In addition, dVols include access control mechanisms to allow tenants to specify restrictions and safeguard against unauthorized accesses and information disclosure.

As the basis for a scalable cloud storage service, Decibel represents a point in the design space that is between host-local ephemeral disks on one hand, and large-scale block storage services such as Amazon's Elastic Block Store (EBS) on the other. Like EBS volumes, dVols are separate from the virtual machines that access them, may be remapped in the face of failure, and allow a greater degree of utilization of storage resources than direct access to local disks. However, unlike volumes in a distributed block store, each dVol in Decibel is stored entirely on a single physical device, provides no storage-level redundancy, and exposes failures directly to the client.

4.3 The Decibel Runtime

Decibel virtualizes the hardware into dVols and multiplexes these dVols onto available physical resources to ensure isolation and to meet their performance objectives. Each Decibel instance is a single-host runtime that is responsible solely for abstracting the remote, shared nature of disaggregated storage from tenants. Decibel can provide ephemeral storage directly to tenants or act as a building block for a larger distributed storage system where multiple Decibel instances are combined to form a network filesystem or an object store [39, 152].

Decibel's architecture is shown in Figure 4.1: it partitions system hardware into independent, shared-nothing per-core runtimes. As achieving efficient resource utilization requires concurrent, wait-free processing of requests and needs to eliminate synchronization and coherence traffic that is detrimental to performance, Decibel opts for full, top-to-bottom system partitioning. Each per-core runtime has exclusive access to a single hardware

queue for every NIC and SCM in the system and can access the hardware without requiring co-ordination across cores.

Decibel relies on kernel-bypass libraries to partition the system and processes I/O traffic within the application itself; a network stack on top of the Intel Data Plane Development Kit (DPDK) [74] and a block layer on top of the Intel Storage Plane Development Kit (SPDK) [77] provide direct low-latency access to network and storage resources within the user space application. Each per-core runtime operates independently and is uniquely addressable across the network. Execution on each core is through a single kernel thread on which the runtime co-operative schedules network and storage I/O and request processing along with virtualization and other datapath services.

Each per-core runtime services requests from multiple tenants for multiple dVols, while each dVol is bound to a single core. The mapping from dVols to the host and core is reflected in the network address directory, which is a separate, global control path network service. As self-contained entities, dVols can be migrated across cores and devices within a host or across hosts in response to changes in load or performance objectives.

By forcing all operations for a dVol to be executed serially on a single core, Decibel avoids the contention overheads that have plagued high-performance concurrent systems [20, 28]. Binding dVols to a single core and SCM restricts the performance of the dVol to that of a single core and device, forcing Decibel to rely on client-side aggregation where higher throughput or greater capacity are required. We anticipate that the runtime can be extended to split existing hot dVols across multiple cores [3] to provide better performance elasticity.

Clients provision and access dVols using a client-side library that maps client interfaces to remote RPCs. The library also handles interaction with the network address directory, allowing applications to remain oblivious to the remote nature of dVols. Legacy applications could be supported through a network block device driver; however, this functionality is currently not provided.

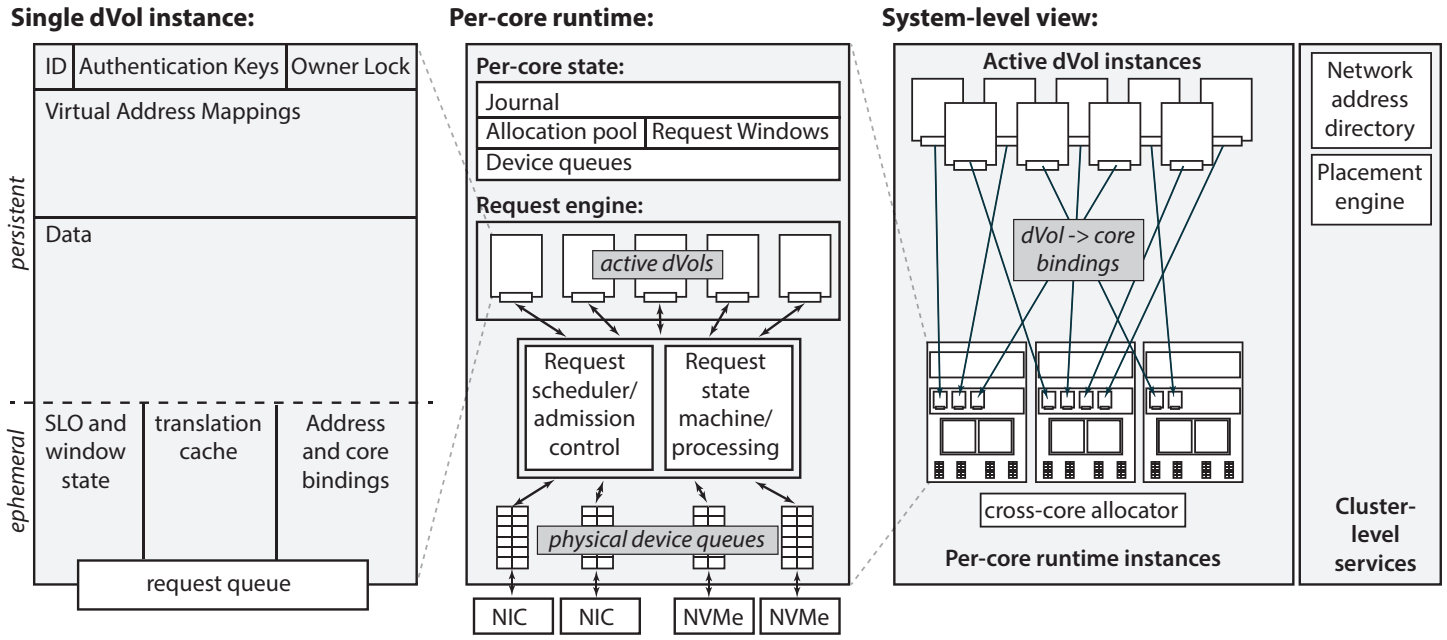

Figure 4.1: dVol and per-core runtime architecture in Decibel

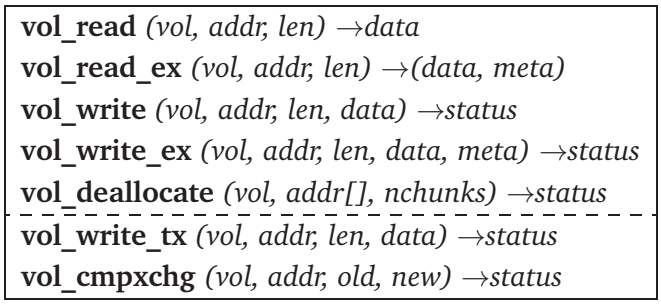

Figure 4.2: Datapath Interfaces for dVols. The last two provide functionality not directly provided by SCMs in hardware.

4.3.1 Virtual Block Devices

Storage virtualization balances the need to preserve the illusion of exclusive, locally-attached disks for tenants with the necessity of supporting operational and management tasks for datacenter operators. The tenant interfaces to virtualized storage, enumerated in Figure 4.2, closely match that of the underlying hardware with commands such as read, write, and deallocate[2] providing the same semantics as the corresponding NVMe commands.

Device Partitioning: dVols provide tenants a sparse virtual address space backed by an SCM. As the storage requirements of tenants vary over time, the capacity utilization of dVols constantly grows and shrinks during execution. Consequently, Decibel must manage the fine-grained allocation of capacity resources across dVols.

One alternative for device partitioning is to rely on hardware-based NVMe namespaces [163] which divide SCMs into virtual partitions that may be presented directly to tenants. As implemented in modern hardware, namespaces represent large contiguous physical regions of the device, making them unsuitable for dynamically resizing workloads. Moreover, many NVMe devices do not support namespaces at all, and where they are supported, devices are typically limited to a very small number[3] of namespace instances.

[2]We use the NVMe "deallocate" command, also termed "trim", "unmap", or "discard" in the context of SATA/SAS SSDs.

[3]The maximum number supported today is 16, with vendors indicated that devices sup-

While the namespace idea is, in principle, an excellent abstraction at the device layer, these limits make them insufficient today, and are one of the reasons that Decibel elects to virtualize the SCM address space above the device itself.

Decibel virtualizes SCMs at block-granularity. Blocks are 4K contiguous regions of the device's physical address space. While some SCMs support variable block sizes, Decibel uses 4K blocks to match both existing storage system designs and x86 memory pages. Blocks are the smallest writeable unit that do not require firmware read-modify-write (RMW) cycles during updates, and also generally the largest unit that can be atomically overwritten by SCMs for crash-safe in-place updates.

Address Virtualization: dVols map the virtual address space presented to tenants onto physical blocks using a private *virtual-to-physical* (V2P) table. Each dVol's V2P table is structured as a persistent B+ tree, with fixed, block-sized internal and leaf nodes, and is keyed by 64-bit virtual addresses; internal nodes store references to their children as 64-bit physical block addresses.

V2P mappings are stored as metadata on the SCM. Client writes are fully persisted, including both data and V2P mappings, before being acknowledged. Decibel performs soft-update-ordered [52] writes of data blocks and metadata: where a write requires an update to the V2P table, data is always written and acknowledged by the device before the associated metadata write is issued. The current implementation is conservative, in that all V2P transactions are isolated. There is opportunity to further improve performance by merging V2P updates. Subsequent writes to allocated blocks do not modify the V2P table and occur in-place, relying on the block-level write atomicity of SCMs for consistency.

Several modern SCMs show little benefit for physically contiguous accesses, especially in multi-tenant scenarios with mixed reads and writes. As a result, dVols do not preserve contiguity from tenant writes and split large, variable-sized requests into multiple, block-sized ones. V2P mappings are

porting 128 namespaces are likely to be available over the next few years.

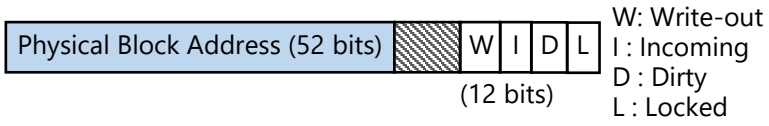

Figure 4.3: Cached V2P Entry

also stored at a fixed, block-sized granularity. This trades request amplification and an increase in V2P entries for simplified system design: Decibel does not require background compaction and defragmentation services, while V2P entries avoid additional metadata for variable-sized mappings.

Decibel aggressively caches the mappings for every dVol in DRAM. The V2P table for a fully-allocated terabyte-sized device occupies approximately 6 GB (an overhead of 0.6%). While non-trivial, this is well within the limits of a high performance server. Cached V2P entries vary from the on-device format: as Figure 4.3 illustrates, physical addresses are block-aligned and require only 52 bits, so the remaining 12 bits are used for entry metadata.

The *Incoming* and *Write-out* bits are used for cache management and signify that the entry is either waiting to be loaded from the SCM or that an updated entry is being flushed to the SCM and is awaiting an acknowledgement for the write. The *Dirty* bit indicates that the underlying data block has been modified and is used to track dirtied blocks to copy during dVol migrations. The *Locked* bit provides mutual exclusion between requests to overlapping regions of the dVol: when set, it restricts all access to the mapping for any request context besides the one that has taken ownership of the lock.

Block Allocation: Requests to allocate blocks require a consistent view of allocations across the entire system to prevent races and double allocations. Decibel amortizes the synchronization overhead of allocations by splitting them into *reservations* and *assignment*: each core reserves a fixed-size, physically-contiguous collection of blocks called an *extent* from the device in a single operation and adds it to a per-core allocation pool (resembling the thread cache in `tcmalloc`).

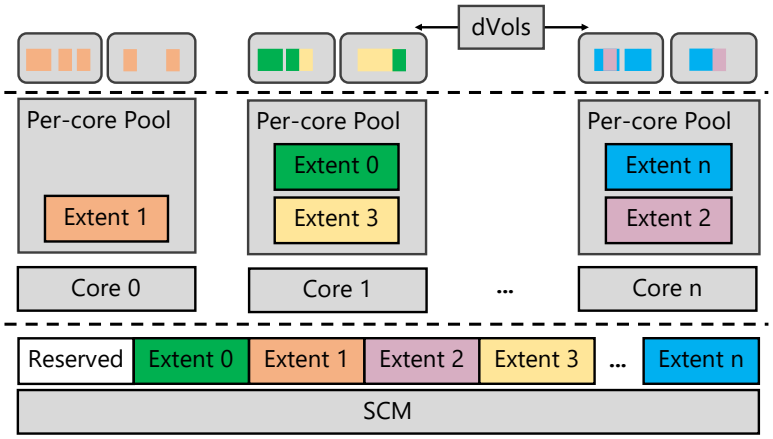

Figure 4.4: Physical Partitioning of Storage Devices

As seen in Figure 4.4, SCMs are divided into multiple extents, which are dynamically reserved by cores. The reservation of extents to cores is tracked by a global, per-device allocator. Cores asynchronously request more extents from the allocator once the number of available blocks in their local pool falls below a certain threshold. This ensures that, as long as the device is not full, allocations succeed without requiring any synchronization.

Assigning entire extents to dVols risks fragmentation and a wastage of space. Instead, cores satisfy allocation requests from dVols by assigning them blocks from any extent in their private pool at a single block granularity. As individual blocks from extents are assigned to different dVols, the split-allocation scheme eliminates both fragmentation and contention along the datapath.

Internally, extents track the allocation status of blocks using a single block-sized bitmap; as every bit in the bitmap represents a block, each extent is 128 MB $(4K \times 4K \times 8)$. Restricting the size of extents to the representation capacity of a single block-sized bitmap allows the bitmap to atomically be overwritten after allocations and frees.

dVols explicitly free blocks that are no longer needed using the deallocate command, while deleting dVols or migrating them across devices im-

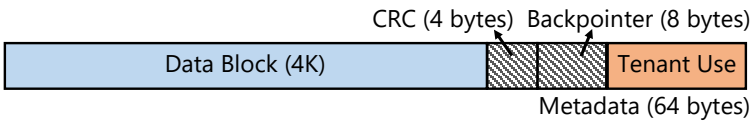

Figure 4.5: Extended Logical Blocks (Block + Metadata)

plicitly triggers block deallocations. Freed blocks are returned to the local pool of the core where they were originally allocated and are used to fulfil subsequent allocation requests.

Data Integrity and Paravirtual Interfaces: SCMs are increasingly prone to block errors and data corruption as they age and approach the endurance limits of flash cells [134]. Even in the absence of hardware failures, SCMs risk data corruption due to *write-tearing*: since most SCMs do not support atomic multi-block updates, failures during these updates result in partial writes that leave blocks in an inconsistent state.

Decibel provides additional services not directly available in hardware to help prevent and detect data corruption. On each write, it calculates block-level checksums and verifies them on reads to detect corrupted blocks before they propagate through the system. dVols also support two additional I/O commands: *multi-block atomicity* to protect against write-tearing and *block compare-and-swap* to allow applications that can only communicate over shared storage to synchronize operations using persistent, on-disk locks [156].

Several enterprise SCMs support storing per-block metadata alongside data blocks and updating the metadata atomically with writes to the data. The block and metadata regions together are called *extended logical blocks* (shown in Figure 4.5). Block metadata corresponds to the Data Integrity Field (DIF) provided by SCSI devices and is intended for use by the storage system. Decibel utilizes this region to store a CRC32-checksum of every block.

Block checksums are self-referential integrity checks that protect against data corruption, but offer no guarantees about metadata integrity, as V2P

entries pointing to stale or incorrect data blocks are not detected. Metadata integrity can be ensured either by storing checksums with the metadata or by storing backpointers alongside the data. To avoid updating the mappings on every write, Decibel stores backpointers in the metadata region of a block. As the data, checksum, and backpointer are updated atomically, Decibel overwrites blocks in-place and still remains crash consistent.

The metadata region is exposed to tenants through the extended, metadata-aware read and write commands (see Figure 4.2) and can be used to store application-specific data, such as version numbers and cryptographic capabilities. As this region is shared between Decibel and tenants, the extended read and write functions mask out the checksum and backpointer before exposing the remainder of the metadata to tenants.

Since most SCMs are unable to guarantee atomicity for writes spanning multiple blocks, Decibel provides atomic updates using block-level copy-on-write semantics. First, new physical blocks are allocated and the data written, following which the corresponding V2P entries are modified to point to the new blocks. Once the updated mappings are persisted, the old blocks are freed. As the V2P entries being updated may span multiple B-tree nodes, a lightweight journal is used to transactionalize the update and ensure crash-consistency.

To perform block-level CAS operations, Decibel first ensures that there are no in-flight requests for the desired block before locking its V2P entry to prevent access until the operation is complete. The entire block is then read into memory and tested against the compare value; if they match, the swap value is written to the block. Storage systems have typically used CAS operations, when available, to co-ordinate accesses to ranges of a shared device or volume without locking the entire device.

Provisioning and Access Control: Access control mechanisms restrict a tenant's view of storage to only the dVols it is permitted to access. Decibel uses a lightweight, token-based authentication scheme for authorization and *does not* protect data confidentiality via encryption, as such CPU-intensive facilities are best left to either the clients or the hardware.

vol_create *() →(vol, token)*
vol_restrict *(vol, type, param) →status*
vol_open *(vol, token) →status*
vol_change_auth *(vol, token) →newtoken*
vol_delete *(vol, token) →status*

Figure 4.6: Provisioning and Access Control Interfaces

Figure 4.6 enumerates the control plane interfaces, presented to tenants, to provision dVols and manage access control policies for them. While creating dVols, Decibel generates a random globally unique identifier and token pair, which are returned to the tenant for use as a volume handle and credentials for future access.

In addition to credential-based authorization, dVols can also restrict access on the basis of network parameters, such as a specific IP address, or to certain VLANs and VXLANs[4]. By forcing all traffic for a dVol onto a private network segment, Decibel allows policies to be applied within the network; for example, traffic can be routed through middleboxes for packet inspection or rely on traffic shaping to prioritize latency-sensitive workloads.

4.3.2 Scheduling Storage

Virtualization allows tenants to operate as if deployed on private, local storage, while still benefiting from the flexibility and economic benefits of device consolidation within the datacenter. For practical deployments, preserving only an interface resembling local storage is insufficient: the storage system must also preserve the performance of the device and insulate tenants from interference due to resource contention and sharing.

The need for performance isolation in multi-tenant storage systems has led to the development of several algorithms and policies to provide fair sharing of devices and guaranteeing tenant throughput [57–59, 85, 104, 141, 142, 153, 157, 158, 167] and for providing hard deadlines for re-

[4]Virtual Extensible LANs (VXLANs) provide support for L2-over-L4 packet tunneling and are used to build private overlay networks.

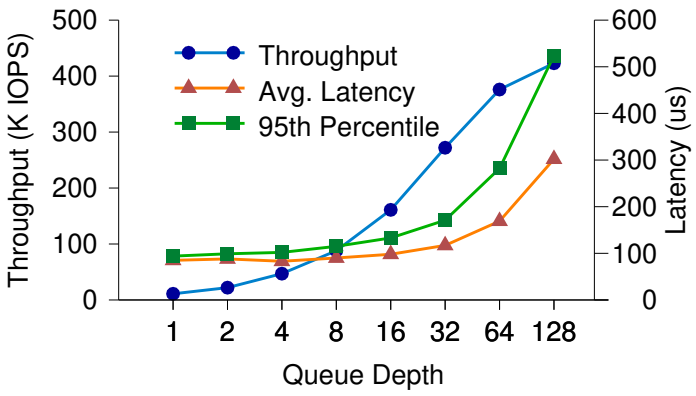

Figure 4.7: Throughput and Latency of Reads

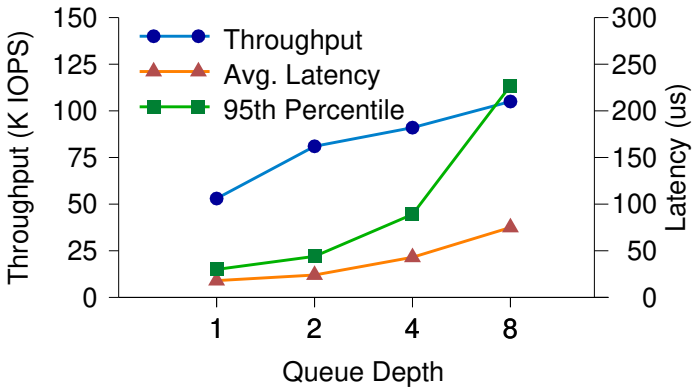

Figure 4.8: Throughput and Latency of Writes

quests [58, 85, 158, 167]. Rather than prescribing a particular policy, Decibel provides dVols as a policy enforcement tool for performance isolation.

SLOs: Throughput and Latency: Figure 4.7 and Figure 4.8 compare the throughput and latency for both reads and writes of a single device, measured locally at different queue depths. The results lead us to two observations about performance isolation for SCMs:

There is a single latency class for the entire device. Even for multi-queue devices, request latency depends on overall device load. Despite the fact that the NVMe specification details multiple scheduling policies across submission queues for QoS purposes, current devices do not sufficiently insulate requests from different queues to support multiple latency classes for a single device. Instead Decibel allows the storage administrator to pick a throughput target for every SCM, called the *device ceiling*, to match a desired latency target.

Providing hard latency guarantees is not possible on today's devices. Comparing average and $95th$ percentile latencies for the device, even at relatively low utilization levels, reveal significant jitter, particularly in the case of writes. Long tail latencies have also been observed for these devices in real deployments [63]. This is largely due to the Flash Translation Layer (FTL), a hardware indirection layer that provides background bookkeeping operations such as wear levelling and garbage collection.

Emerging hardware that provides predictable performance by managing flash bookkeeping in software is discussed in Section 4.5. In the absence of predictable SCMs, Decibel focuses on preserving device throughput. dVols encapsulate an SLO describing their performance requirements, either in terms of a proportional share of the device or a precise throughput target.

Characterizing Request Cost: Guaranteeing throughput requires the scheduler to be able to account for the cost of every request before deciding whether to issue it to the device. Request costs, however, are variable and a function of the size and nature of the request, as well as the current load on the SCM. For example, writes are significantly cheaper than reads as long as they are being absorbed by the SCM write buffer, but become much more expensive once the write buffer is exhausted.

The Decibel scheduler does not need to account for variable-sized tenant requests as the address translation layer of the dVol converts them into uniform 4K requests at the block layer. As a simplifying assumption, the scheduler does not try and quantify the relative costs of reads and writes, but instead requires both the device ceiling and SLOs to specify read and

write targets separately. In the future, we intend to extend the scheduler to provide a unified cost model for reads and writes [141].

Request Windows: At a specified device ceiling, Decibel determines the size of the global request window for an SCM, or the total number of outstanding requests that can be issued against the device. Each per-core runtime has a private request window, where the sizes of all the individual request windows are equal to that of the global request window for the device. The size of the private request window for cores is calculated on the basis of the SLO requirements of dVols scheduled to execute on that core. As dVols are created, opened, moved, or destroyed, Decibel recalculates the private window sizes, which are periodically fetched by the cores.

dVols submit requests to devices by enqueuing them in private software queues. While submitting requests to the device, the per-core runtime selects requests from the individual dVol queues until either the request window is full or there are no pending requests awaiting submission. The runtime chooses requests from multiple dVols on the basis of several factors, such as the scheduling policy, the dVol's performance requirements, and how many requests the dVols have submitted recently.

Execution Model: Per-core runtimes co-operatively schedule dVols on a single OS thread: the request processor issues asynchronous versions of blocking syscalls and yields in a timely manner. Decibel polls NICs and SCMs on the same thread to eliminate context switching overheads and to allow the schedulers to precisely control the distribution of compute cycles between servicing the hardware and the request processing within dVols.

Request processing within the dVol includes resolving virtualization mappings and performing access control checks; consequently, requests may block and yield several times during execution and cannot be run to completion as in many memory-backed systems. The scheduler treats requests and dVols as analagous to threads and processes—scheduling operates at the request level on the basis of policies applying to the entire dVol. At any given time the scheduler dynamically selects between executing the net-

work or storage stacks, and processing one of several executable requests from multiple dVols.

Storage workloads are bursty and susceptible to incast [127]; as a result, Decibel is periodically subject to bursts of heavy traffic. At these times, the Decibel scheduler elastically steals cycles to prioritize handling network traffic. It polls NICs at increased frequencies to ensure that packets are not dropped due to insufficient hardware buffering, and processes incoming packets just enough to generate ACKs and prevent retransmissions.

Prioritizing network I/O at the cost of request processing may cause memory pressure due to an increase in the number of pending requests. At a certain threshold, Decibel switches back to processing requests, even at the cost of dropped packets. As dropped packets are interpreted as network congestion, they force the sender to back-off, thus inducing back pressure in the system.

Scheduling Policies: The scheduling policy determines how the per-core runtime selects requests from multiple dVols to submit to the device. To demonstrate the policy-agnostic nature of Decibel's architecture, we prototype two different scheduling policies for dVols.

Strict Time Sharing (STS) emulates local storage by statically partitioning and assigning resources to tenants. It sacrifices elasticity and the ability to handle bursts for more predictable request latency. Each dVol is assigned a fixed request quota per scheduling epoch from which the scheduler selects requests to submit to the device. dVols cannot exceed their quota even in the absence of any competition. Further, dVols do not gain any credits during periods of low activity, as unused quota slots are not carried forward.

Deficit Round Robin (DRR) [140] is a work conserving scheduler that supports bursty tenant access patterns. DRR guarantees that each dVol is able to issue its fair share of requests to the device, but does not limit a dVol to only its fair share in the absence of competing dVols. Each dVol has an assigned quota per scheduling epoch; however, dVols that do not use their entire quota carry it forward for future epochs. As dVols can exceed their quota in the absence of competition, bursty workloads can be accommo-

dated.

By default, Decibel is configured with DRR to preserve the flexibility benefits of disaggregated storage. This remains a configurable parameter to allow storage administrators to pick the appropriate policies for their tenants.

4.3.3 Placement and Discovery

Decibel makes the decision to explicitly decouple *scheduling* from the *placement* of dVols on the appropriate cores and hosts in the cluster. This division of responsibilities allows the scheduler to focus exclusively on, as seen earlier, providing fine-grained performance isolation and predictable performance over microsecond timeframes on a per-core basis. Placement decisions are made with a view of the cluster over longer timeframes in response to the changing capacity and performance requirements of dVols. Decibel defers placement decisions to an external placement engine called Mirador [161]. Mirador is a global controller that provides continuous and dynamic improvements to dVol placements by migrating them across cores, devices, and hosts and plays a role analagous to that of an SDN controller for network flows.

Storage workloads are impacted by more than just the local device; for example, network and PCIe bandwidth oversubscription can significantly impact tenant performance. Dynamic placement with global resource visibility is a response to not just changing tenant requirements, but also to connectivity bottlenecks within the datacenter. Dynamic dVol migrations, however, raise questions about how tenants locate and access their dVols.

dVol Discovery: Decibel implements a global directory service that maps dVol identifiers to the precise host and core on which they run. Cores are independent network-addressable entities with a unique `<ip:port>` identifier and can directly be addressed by tenants. The demultiplexing of tenant requests to the appropriate dVol happens at the core on the basis of the dVol identifier.

dVol Migration: The placement engine triggers migrations in response to capacity or performance shortages and aims to find a placement schedule that ensures that both dVol SLOs are met and that the free capacity of the cluster is uniformly distributed across Decibel instances to allow every dVol an opportunity to grow. Migrations can be across cores on the same host or across devices within the same or different hosts.

Core migrations occur entirely within a single Decibel instance. dVols are migrated to another core within the same host without requiring any data movement. First, Decibel flushes all the device queues and waits for in-flight requests to be completed, but no new dVol requests are admitted. The dVol metadata is then moved to the destination core and the address directory updated. The client is instructed to invalidate its directory cache and the connection is terminated; the client then connects to the new runtime instance and resumes operations.

Device migrations resemble virtual machine migration and involve a background copy of dVol data. As the dVol continues to service requests, modified data blocks are tracked using the dirty bit in the V2P table and copied to the destination. When both copies approach convergence, the client is redirected to the new destination using the same technique as core migrations and the remaining modified blocks moved in a post-copy pass.

Decibel originally intended to perform migrations without any client involvement using OpenFlow-based redirection in the network and hardware flow steering at the end-hosts. Due to the limited availability of match rules at both switches and NICs today, Decibel opts to defer this functionality to the client library.

4.3.4 The Network Layer

Decibel presents the dVol interface over asynchronous TCP/IP-based RPC messages. Network flows are processed using a user space networking stack that borrows the TCP state machine and structures for processing TCP flows, such as the socket and flow tables, from mTCP [80] and combines them with custom buffer management and event notification systems. Decibel

offloads checksums to the hardware, but currently does not support TCP segmentation offload.

Clients discover core mappings of dVols using the network address directory. As dVols are pinned to cores that have exclusive ownership over them, tenants must direct requests to the appropriate core on the system. Modern NICs provide the ability to precisely match specific fields in the packet header with user-defined predicates and determine the destination queue for that packet on the basis of provided rules. As each core has a unique `<ip:port>`, Decibel uses such flow steering to distribute incoming requests across cores directly in hardware.

For performance reasons, Decibel extends the shared-nothing architecture into the networking layer. It borrows ideas from scalable user space network stacks [12, 80, 105, 125, 131] and partitions the socket and flow tables into local, per-core structures that can be accessed and updated without synchronization.

Memory Management: Decibel pre-allocates large per-core regions of memory for sockets, flow tables, and socket buffers from regular memory and `mbufs` for network packets from hugepages. `mbufs` are stored in lockless, per-socket send and receive ring buffers; the latter is passed to DPDK which uses them to DMA incoming packets. Decibel does not support zero-copy I/O: incoming packet payloads are staged in the receive socket buffer for unpacking by the RPC layer, while writes are buffered in the send socket buffer before transmission. Zero-copy shows little benefit on processors with Direct Data I/O (DDIO), i.e., the ability to DMA directly into cache [105]. Further, once packets in `mbufs` are sent to the DPDK for transmission, they are automatically freed and unavailable for retransmissions, making zero-copy hard without complicating the programming interface.

Event Notifications and Timers: As request processing and the network stack execute on the same thread, notifications are processed in-line via callbacks. Decibel registers callbacks for new connections and for I/O events: `tcp_accept()`, `tcp_rcv_ready()`, and `tcp_snd_ready()`. Send and

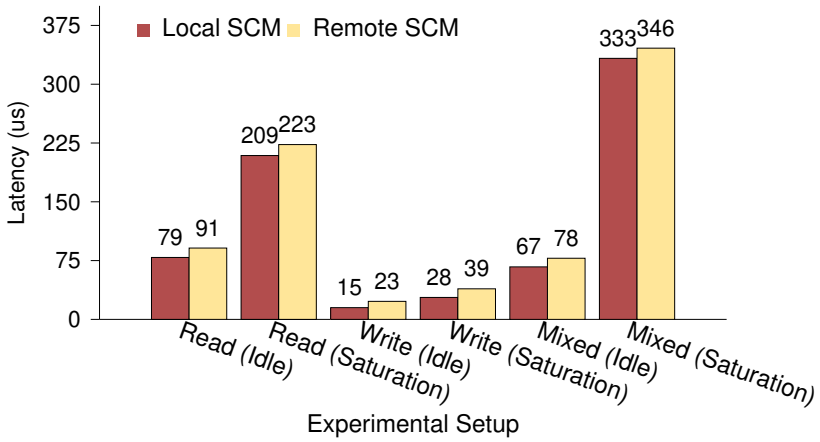

Figure 4.9: Local and Remote Latency for 4K Requests

receive callbacks are analagous to `EPOLLIN` and `EPOLLOUT` and signify the availability of data or the ability to transmit data. The callbacks themselves do not carry any data but are only event notifications for the request processor to act on using the socket layer interfaces. Timers for flow retransmissions and connection timeouts, similarly, cannot rely on external threads or kernel interrupts to fire and instead are tracked using a hashed timer wheel and processed in-line along with other event notifications.

Why not just use RDMA? Several recent high-performance systems have exploited RDMA (Remote Direct Memory Access)—a hardware mechanism that allows direct access to remote memory without software mediation—to eliminate network overheads and construct a low-latency communication channel between servers within a datacenter, in order to accelerate network services, such as key-value stores [44, 81, 111] and data parallel frameworks [61, 78].

RDMA's advantage over traditional networking shrinks as request sizes grow [81, 111], especially in the presence of low-latency, kernel-bypass I/O libraries. Figure 4.9 compares local and remote access latencies, over TCP/IP-based messaging, for SCMs when they are relatively idle (for min-

imum latencies) and at saturation. For a typical storage workload request size of 4K, conventional messaging adds little overhead to local accesses.

RDMA has traditionally been deployed on Infiniband and requires lossless networks for performance, making it hard to incorporate into existing Ethernet deployments. On Ethernet, RDMA requires an external control plane to guarantee packet delivery and ordering [61] and for congestion control to ensure link fairness [168].

Decibel's choice of traditional Ethernet-based messaging is pragmatic, as the advantages of RDMA for storage workloads do not yet outweigh the significant deployment overheads. As RDMA-based deployments increase in popularity, and the control plane protocols for prioritizing traffic and handling congested and lossy networks are refined, this may no longer hold true. Consequently, Decibel's architecture is mostly agnostic to the messaging layer and is capable of switching to RDMA if required by the performance of future SCMs.

4.4 Evaluation

Decibel is evaluated on a pair of 32-core Haswell systems, each with 2x40 GbE X710 NICs and 4x800 GB P3700 NVMe PCIe SSDs, with one system acting as the server and the other hosting multiple clients. Each machine has 64 GB RAM split across two NUMA nodes, while the 40 GbE interfaces are connected via an Arista 7050 series switch. Both systems run a Linux 4.2 kernel, however, on the server Decibel takes exclusive ownership of both the network and storage adapters. Clients are measured both using the default kernel I/O stack and the DPDK-based network stack from Decibel.

At 4K request sizes, each P3700 is capable of up to 460K random read IOPS, 100K random write IOPS, and 200K random mixed (at a 70/30 read to write ratio) IOPS [76], making the saturated throughput of the system up to 1.8M read IOPS and 800K mixed IOPS. Benchmarking flash-based SSDs is non-trivial as there are a number of factors that may affect their performance. First, the performance of a new SSD is not indicative of how it would perform at *steady state* with fresh drives outperforming their steady

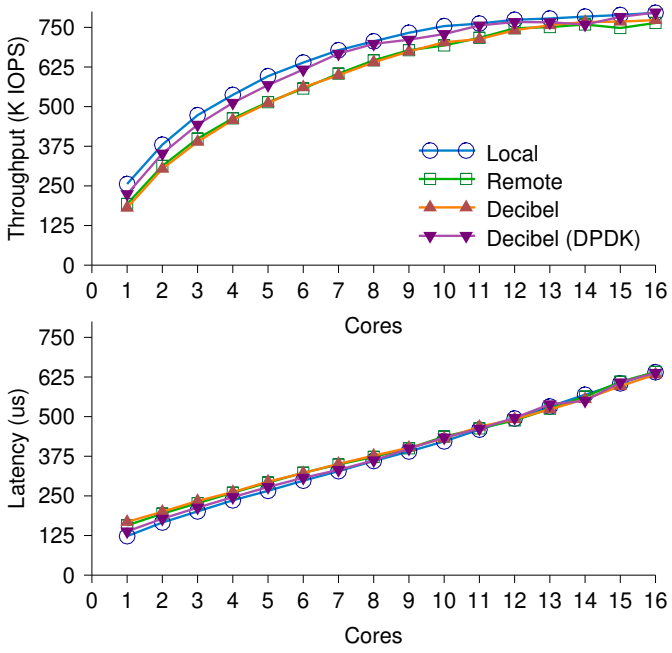

Figure 4.10: Performance of Decibel for a 70/30 read-write workload. Compared to local storage, Decibel has an overhead of 30 μs at device saturation using a DPDK-based client.

state counterparts by a factor or two or three.

Even once steady state is reached, there is a great deal of variability in performance. Transitions from sequential to random and vice versa impact performance for several minutes, while the garbage collector can throttle disk throughput for several seconds. The P3700s, in particular, perform well past their rated write throughput for almost a minute following a period of idleness [130]. The results reported here are the average across a 10 minute run and follow industry standard guidelines for benchmarking [143]: first the devices were pre-conditioned with several weeks of heavy usage and then primed by running the same workload access pattern as the benchmark for 10 minutes prior to the benchmark run.

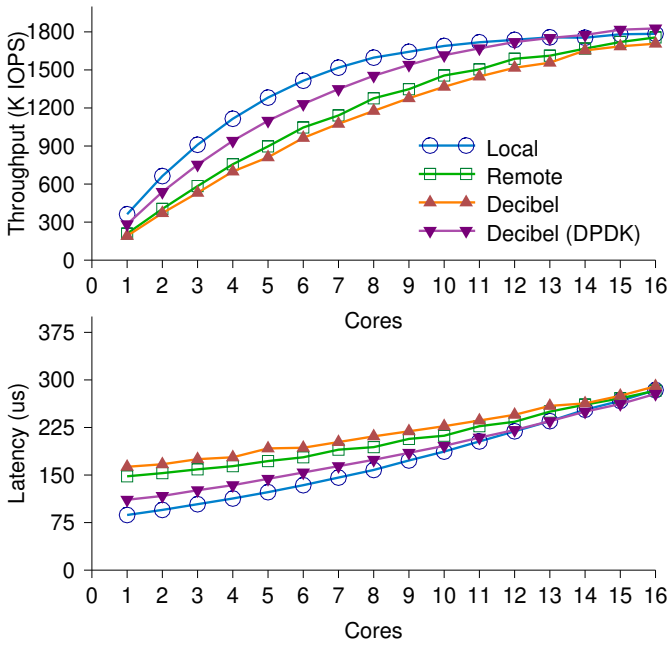

Figure 4.11: Performance of Decibel for an all reads workload. Compared to local storage, Decibel has an overhead of less than 20 μs at device saturation using a DPDK-based client.

Remote Overhead and Scalability: Decibel is evaluated for multi-core scalability and to quantify the overhead of disaggregating SCMs when compared to direct-attached storage. All the tests are run against all 4 devices in the system, with clients evenly distributed across the cores. The clients are modelled after fio and access blocks randomly across the entire address space. Local clients execute as a single pinned client per-core with a queue depth of 32, while there are 2 remote clients per-core, each operating with a queue depth of 16 requests.

In the *Local* configuration, clients run directly on the server and access raw physical blocks from the SCMs without any virtualization. This local configuration serves as a baseline to compare the overhead of Decibel. In *Remote*, clients run separately from the server and request raw physical blocks

across the network over TCP/IP. For *Decibel*, SCMs are virtualized into per-client dVols. Each client has a single dVol that is populated until the SCM is filled, after which they access and update blocks within the dVol. The remote configuration measures pure network overhead when compared to directly-attached SCMs, as well as the overhead of virtualization when compared to Decibel. The *Decibel (DPDK)* configuration is identical to Decibel, except that the clients bypass the kernel and use a DPDK-based network stack.

Figure 4.10 compares the performance of all four configurations over 16 cores using a typical storage workload of random mixed 4K requests in a 70/30 read-write ratio. As device saturation is achieved, we do not evaluate Decibel at higher degrees of parallelism.

Decibel is highly scalable and is able to saturate all the devices, while presenting storage across the network with latencies comparable to local storage. DPDK-based clients suffer from an overhead of less than 30 μs when compared to local storage, while legacy clients have overheads varying from 30–60 μs depending on load.

SCMs offer substantially higher throughput for read-only workloads compared to mixed ones making them more heavily CPU-bound. Figure 4.11 demonstrates Decibel's ability to saturate all the devices for read-only workloads: the increased CPU load of processing requests is reflected in the gap with the local workload at low core counts. As the number of cores increase, the workload becomes SCM-bound; Decibel scales well and is able to saturate all the devices. At saturation, the DPDK-based client has an overhead of less than 20 μs, while legacy clients suffer from overheads of approximately 90 μs.

Once the devices are saturated, adding clients increases latency purely due to queueing delays in software. All configurations saturate the devices at less than 16 cores; hence the latency plots in Figure 4.10 and Figure 4.11 include queueing delays and do not accurately reflect end-to-end latencies. Table 4.2 compares latencies at the point of device saturation: for both workloads, Decibel imposes an overhead of 20–30 μs on DPDK-based clients compared to local storage.

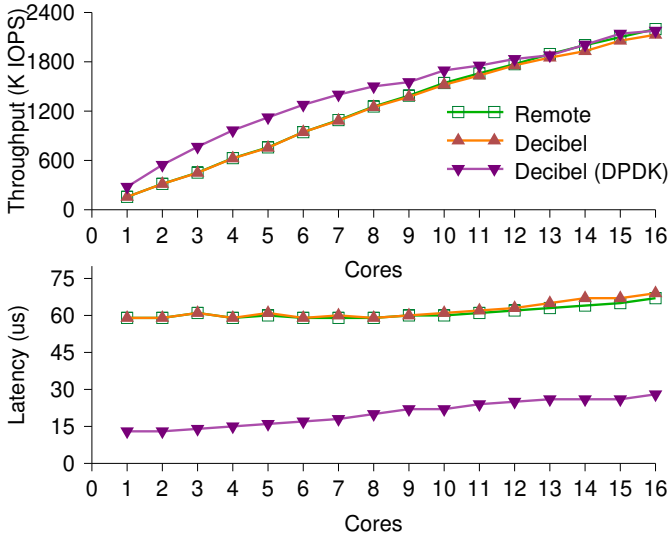

Figure 4.12: Remote access latencies for Decibel at different degrees of device utilization against DRAM-backed storage.

	70/30		**All Reads**	
	Xput	Lat	Xput	Lat
Local	750K	422	1.7M	203
Remote	740K	488	1.7M	283
Decibel	740K	490	1.7M	290
Decibel (DPDK)	750K	450	1.7M	221

Table 4.2: Performance for Workloads (Latency in μs)

Future SCMs, such as 3DXpoint [115], are expected to offer sub-μs latencies for persistent memories and around 10 μs latencies for NVMe storage. With a view towards these devices, we evaluate Decibel against a DRAM-backed block device. As seen in Figure 4.12, DPDK-based clients have access latencies of 12–15 μs at moderate load, which increases to 26 μs at NIC saturation. Legacy clients have access latencies higher than 60 μs, demonstrating that the kernel stack is a poor fit for rack-scale storage architectures.

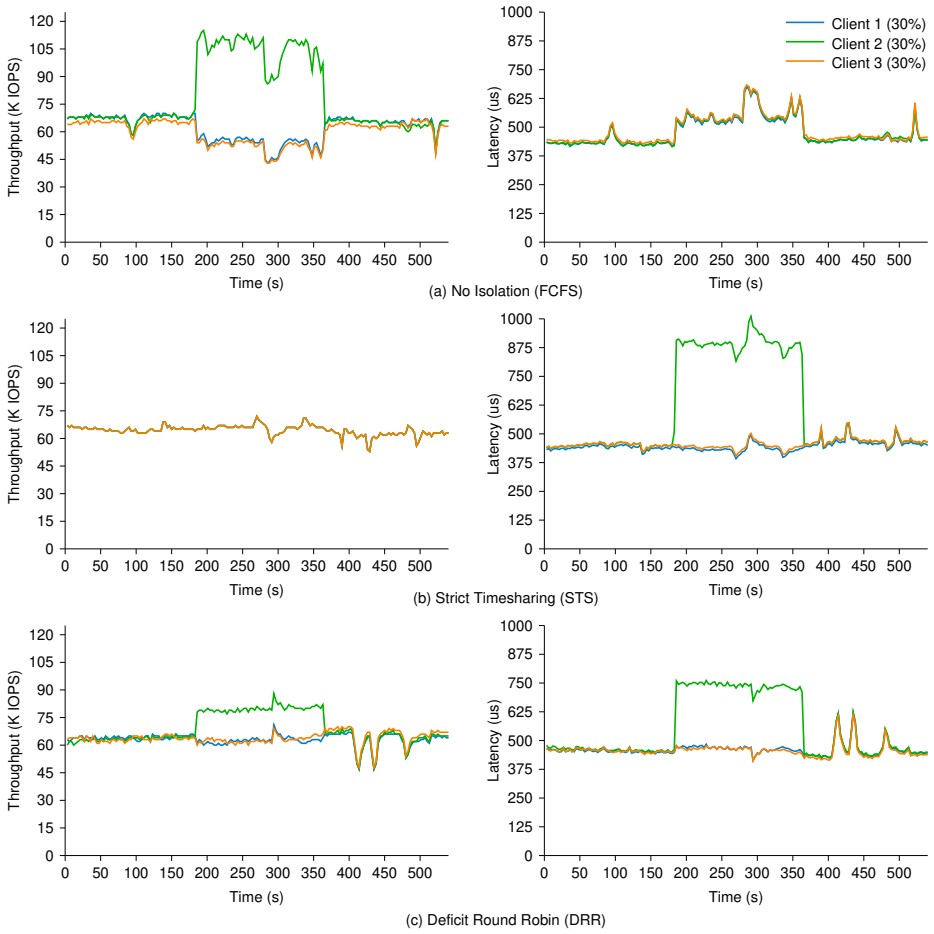

Figure 4.13: Isolation of a single device across multiple workloads in Decibel. Compared to the no isolation case in (a), the scheduling policies in (b) and (c), provide clients 1 and 3 a fair share of the device, even in the face of the bursty accesses of client 2.

dVol Isolation: Performance isolation in Decibel is evaluated by demonstrating fair sharing of a device in two different scheduling policies when compared with a First-Come, First-Served (FCFS) scheduler that provides no performance isolation. Strict Timesharing (STS) provides static resource partitioning, while in Deficit Round Robin (DRR), dVols are prevented from

interfering with the performance of other dVols, but are allowed to consume excess, unused bandwidth.

To illustrate performance isolation, Decibel is evaluated with three clients, each with a 70/30 mixed random workload, against a single shared SCM. Each client continuously issues requests to its own dVol, such that the dVol has 30 outstanding requests at any time. The dVols are configured to have an equal proportion, i.e., 30%, of the total device throughput, while the device ceiling is set to 100% utilization.

As seen in Figure 4.13, each client receives a throughput of 60K, leaving the device at 90% saturation. At the 3 minute mark, one of the clients experiences a traffic burst for 3 minutes such that it has 90 simultaneous in-flight requests. At 6 minutes, the burst subsides and the client returns to its original load.

FCFS offers no performance isolation, allowing the burst to create queueing overheads which impact throughput and latency of all other clients by 25%. After the burst subsides, performance returns to its original level. In contrast, STS preserves the throughput of all the clients and prevents clients from issuing any requests beyond their 30% reservation. As each dVol has hard reservations on the number of requests it can issue, requests from the bursty client are queued in software and see huge spikes in latency. The performance of the other clients remains unaffected at all times, but the excess capacity of the device remains unutilized.

DRR both guarantees the throughput of other clients and is work conserving: the bursty client consumes the unused bandwidth until the device ceiling is reached, but not at the cost of the throughput of other clients. Latency, however, for all the clients rises slightly—this is not because of queueing delays, but because the device latency increases as it gets closer to saturation.

4.5 Related Work

Network-Attached Storage: The idea of centralizing storage in consolidated arrays and exporting disks over the network [86] is not a new one and

has periodically been explored with changes in the relative performance of CPU, networks, and disks. For spinning disk based systems, Petal [96] is a virtualized block store that acts as a building block for a distributed file system [152]. While Petal focuses on aggregating physical disks for performance, Decibel is concerned with the performance challenges of building isolated volumes for SCMs.

More recently, network-attached storage for flash devices has been used in Corfu [7] and Strata [39]. Corfu presents a distributed log over virtualized flash block-devices while storing address translations at the clients. As the clients are trusted, co-operating entities, Corfu does not attempt to provide isolation between them. Strata focuses on providing a global address space for a scalable network file system on top of network-attached storage, and discusses the challenges in providing data plane services such as device aggregation, fault tolerance, and skew mitigation in a distributed manner. In contrast, Decibel is an example of the high-performance network-attached storage such file systems rely on, and provides the services required for multi-tenancy that cannot safely be implemented higher up the stack.

Network-attached Secure Disks (NASD) [53] explore security primitives and capabilities to allow sharing storage devices without requiring security checks at an external file manager on every request, while Snapdragon [2] uses self-describing capabilities to verify requests and limit the blocks a remote client has access to. SNAD [108] performs both tenant authentication and block encryption at the storage server to restrict unauthorized accesses.

Partitioned Data Stores: VoltDB [146] and MICA [99] are both examples of shared-nothing in-memory data stores, which explore vertical partitioning of hardware resources to allow all operations to proceed without expensive cross-core coordination. Architecturally, the per-core runtimes in Decibel resemble those in these systems with the addition of persistent storage devices and the associated datapath services.

Chronos [83] is a more general framework for partitioning applications by running several independent instances in parallel, fronted by a load balancer aware of the instance to partitioning mapping that can route requests

accordingly.

Application Managed Flash: Several recent research storage systems have proposed using open-channel SSDs for more predictable performance [19, 72, 97, 138]. These devices expose internal flash channels, dies, and planes to the system and allow for application-managed software FTLs and custom bookkeeping policies. Of these systems, Flashblox has demonstrated that providing strong isolation and supporting multiple latency classes on a shared SCM requires extending full system partitioning to within the device. By binding device channels and dies directly to tenants in hardware and providing per-tenant accounting for garbage collection, it removes multiple sources of performance interference and maintains low tail latencies in the face of competing tenants.

Application-managed flash is largely complementary to Decibel and focuses largely on providing better and more flexible implementations of services currently provided by the FTL. These systems intentionally maintain a familiar block-like presentation for convenience and, as such, Decibel could integrate with such systems to provide strong end-to-end performance isolation.

4.6 Conclusion

SCMs represent orders of magnitude changes to the throughput, latency, and density of datacenter storage, and have caused a reconsideration in how storage is presented, managed, and accessed within modern datacenters. Decibel responds to the performance realities of SCMs by providing dVols to delegate storage to tenants within fully disaggregated storage architectures. dVols focus exclusively on virtualizing storage and isolating multiple tenants while ensuring that the storage is accompanied with a committed amount of compute and network resources to provide tenants with predictable, low-latency access to data.

Chapter 5

Conclusion

This thesis describes how a software platform can share resources across multiple parties, whether applications, virtual machines, or threads, without sacrificing isolation by mediating all accesses to the underlying hardware. Across the individual systems, which focus on different resources, a number of common themes appear.

The Degree of Isolation: Isolation is, in itself, an amorphous term; consequently, it is incumbent on the designers of the system to define the degree of isolation they aim to provide. The systems described here are pragmatic and restrict isolation to a level which can realistically be provided by the platform without impacting performance. While all three systems significantly increase isolation when compared to existing systems, they fall short of the platonic ideal of complete resource isolation. This is largely due to a lack of support from the underlying hardware, which makes providing a greater degree of isolation extremely expensive.

Xoar increases isolation between virtual machines by securing the shared virtualization platform and by restoring shared components to known-good states between accesses from different tenants. It does not extend its isolation to shared caches as that requires static partitioning and results in poor cache utilization; consequently, virtual machines remain vulnerable to side-channel attacks through the shared cache.

Plastic provides byte-granularity memory virtualization and isolates independent locations that happen to be co-located on a single cache line. Since providing sub-byte granularity isolation, as in the case of independent bits in a bitfield, requires individual bytes to be divided and much heavier weight mediation on every memory access, it quickly becomes prohibitive and is not supported in Plastic.

Decibel isolates storage and is able to provide throughput guarantees for tenants, but does not provide them any latency guarantees. This is because the underlying jitter in storage devices requires that the device be operating well below peak load and sacrificing significant amounts of throughput to meet tight deadlines on tail latencies.

Lightweight Abstractions: Software-based interposition allows the platform to extend hardware and provide isolated resource abstractions on top of devices that provide no facilities for isolation. In providing isolation, however, the platform should not lose sight of the fact that the rationale behind resource sharing in the first place is the performance of these devices, and must not sacrifice performance.

For this reason, the thesis advocates for lightweight resource abstractions and takes a minimalistic approach in their design. Only functionality that is required for isolation and is difficult to provide elsewhere is offered by the abstraction, while all other functionality is deferred to either higher layers of the stack or the application. This allows applications to opt-in to the functionality they desire and layer it on top of the base abstraction, instead of requiring every application to pay the cost of that additional functionality. Decibel is the clearest example of this: by eschewing traditional storage functionality, such as device aggregation and fault-tolerance, it provides near-device levels of performance. Xoar and Plastic leverage existing lightweight abstractions (VMs and threads respectively) and augment them to provide better isolation.

The Evolution of Hardware: Most software systems are point in time solutions designed with the limitations of the prevalent hardware in mind. As both hardware and the challenges faced by systems evolve constantly, the systems techniques required to address them must change accordingly.

Xoar deconstructs the virtualization platform to enable better reasoning about shared software components. Recent hardware trends, however, have reduced the responsibility of the virtualization platform by providing direct hardware virtualization for I/O devices, interrupt handling, and timers. Direct I/O virtualization has another benefit: it makes microrebooting shared driver domains, which is incredibly challenging at the speed of modern I/O devices, unnecessary.

Plastic is a composite of two independent systems: a low-overhead contention detector and a fine-grained address virtualization engine. Contention detection in Plastic took a pipelined approach because the performance counters did not record everything necessary to precisely identify regions of contended memory. On more recent microarchitectures, this is no longer true, allowing false sharing to be precisely identified from the execution traces accompanying contention misses. In retrospect, false sharing is quite rare in production software and is perhaps not the best example to demonstrate the value of runtime optimization of memory layouts. This does not invalidate the idea of continuous monitoring and optimistic optimization, as contention and other memory bottlenecks remain the greatest challenge to multi-core scalability.

With I/O devices continuing to get faster, it is unclear if maintaining software mediation without affecting performance, like in Decibel, will remain possible. Rather than obviating the need for isolation, however, faster hardware will require even greater degrees of sharing to drive efficient utilization. Hybrid architectures that combine hardware-enforced isolation on every access with a software-based control plane for fine-grained control of the enforcement mechanism could make it possible to bypass the processor for most accesses without sacrificing isolation. Such hybrid architectures, which require hardware-software co-design, provide one possible avenue forward for platforms and are an exciting area for future research.

5.1 The Future

The design of the systems described in this thesis was constrained by certain assumptions about the prevailing computing landscape. We revisit these assumptions in light of the evolution in both hardware and software requirements over the last several years, with an eye to discussing interesting directions for further exploration.

The Tyranny of Legacy: All of Xoar, Plastic, and Decibel were designed with legacy in mind and support existing applications with little to no modifications. This was driven by the view that for a system to gain traction and be practically deployable, it should not require additional effort from application developers. In reality, this has turned out to be a largely pessimistic stance: the success of paradigms such as MapReduce [41] and Function-as-a-Service demonstrate the willingness of developers to completely rewrite applications for a new programming model, assuming it provides sufficient benefits in terms of simplicity or cost. Systems designed ground-up with data centers and the cloud in mind should carefully weigh the costs of legacy support and utilize the opportunity to make a clean break from abstractions and interfaces that are poor fit and impede scalability and performance or security and isolation.

Generality and Specialization: Historically, systems abstractions and optimizations have been required to demonstrate generality by evaluating them against a set of "representative" benchmarks such as those from SPEC. This is largely a legacy of the large shared multi-user mainframes of the 80s and 90s; as multiple applications ran on the same OS instance, maximizing aggregate performance across dissimilar applications was a worthy goal.

Modern deployment models are much more specialized with applications often running on bare metal or being hosted in virtual machines and containers with private OS instances. Despite this, the vast majority of recent systems, including Xoar and Plastic, continue to be built and evaluated with generality in mind. This generality comes at a cost. As a significant pro-

portion of total data center computation involves a small set of application classes, such as data stores and databases, graph computation frameworks, and machine learning toolkits, specializing systems for these specific applications is an increasingly attractive opportunity for future systems.

The Ability to Affect Change in Hardware: As processor fabrication technologies are reaching the limits of silicon semiconductors, processor vendors are increasingly relying on instruction set features, rather than performance improvements, to drive demand [10]. The addition of features to the instruction set is largely a function of the demands of software; for example, several virtualization extensions have been added over the years in response to the challenges faced by hypervisors in cloud environments, while trusted computing extensions are to better support secure third-party hosting and digital rights management. This is not limited to processors: the development of programmable switches and flash devices, both of which allow customization of several previously fixed bits of the I/O data path, is also due to the demands of data center operators for greater flexibility, while Tensor Processing Units (TPUs) are a response to machine learning workloads.

These are hugely exciting developments for systems researchers—never before have they had so much input in the direction of new hardware! Further, they also have much greater access to programmable hardware for cheap prototyping, such as FPGAs. This combination affords them the opportunity to design and prototype systems that split functionality between hardware and software and, for the first time, the ability to see these prototypes through to production should they demonstrate their benefit.

Bibliography

[1] *Department of Defense Trusted Computer System Evaluation Criteria*. DoD 5200.28-STD. U.S. Department of Defense, December 1985. → page 11

[2] AGUILERA, M. K., JI, M., LILLIBRIDGE, M., MACCORMICK, J., OERTLI, E., ANDERSEN, D., BURROWS, M., MANN, T., AND THEKKATH, C. A. Block-Level Security for Network-Attached Disks. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), FAST'03. → pages 83, 112

[3] AGUILERA, M. K., LENERS, J. B., AND WALFISH, M. Yesquel: Scalable SQL Storage for Web Applications. In *Proceedings of the 25th ACM SIGOPS Symposium on Operating Systems Principles* (2015), SOSP'15. → page 88

[4] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., AND VASUDEVAN, V. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), SOSP'09. → page 7

[5] AVAGO. ExpressFabric: Thinking Outside the Box. `http://www.avagotech.com/applications/datacenters/expressfabric`. → pages 6, 83

[6] BAKER, M., AND SULLIVAN, M. The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment. In *Proceedings of the 1992 USENIX Summer Technical Conference* (1992), USENIX'92. → pages 10, 24

[7] BALAKRISHNAN, M., MALKHI, D., PRABHAKARAN, V., WOBBER, T., WEI, M., AND DAVIS, J. D. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th USENIX Conference on Networked*

Systems Design and Implementation (2012), NSDI'12. → pages 83, 112

[8] BARHAM, P., DRAGOVIC, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM SIGOPS Symposium on Operating Systems Principles* (2003), SOSP'03. → pages 8, 54

[9] BARROSO, L. A., AND HÖLZLE, U. The Case for Energy-Proportional Computing. *IEEE Computer 40*, 12 (December 2007). → page 7

[10] BAUMANN, A. Hardware is the new Software. In *Proceedings of the 16th USENIX Conference on Hot Topics in Operating Systems* (2017), HotOS'17. → page 118

[11] BAUMANN, A., BARHAM, P., DAGAND, P.-E., HARRIS, T., ISAACS, R., PETER, S., ROSCOE, T., SCHÜPBACH, A., AND SINGHANIA, A. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), SOSP'09. → page 50

[12] BELAY, A., PREKAS, G., KLIMOVIC, A., GROSSMAN, S., KOZYRAKIS, C., AND BUGNION, E. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14. → page 103

[13] BELLARD, F. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference* (2005), ATC'05. → page 14

[14] BERGAN, T., HUNT, N., CEZE, L., AND GRIBBLE, S. D. Deterministic Process Groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10. → page 60

[15] BERNAT, A. R., AND MILLER, B. P. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN Workshop on Program Analysis for Software Tools* (2011), PASTE'11. → page 63

[16] BIENIA, C., AND LI, K. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation* (June 2009). → page 77

[17] BITTAU, A., MARCHENKO, P., HANDLEY, M., AND KARP, B. Wedge: Splitting Applications into Reduced-privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), NSDI'08. → page 44

[18] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. In *Proceedings of the 6th International Systems and Storage Conference* (2013), SYSTOR'13. → page 84

[19] BJØRLING, M., GONZALEZ, J., AND BONNET, P. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17. → page 113

[20] BOLOSKY, W. J., AND SCOTT, M. L. False Sharing and Its Effect on Shared Memory Performance. In *Proceedings of the 4th USENIX Symposium on Experiences with Distributed and Multiprocessor Systems* (1993), SEDMS'93. → pages 53, 88

[21] BOYD-WICKIZER, S., CLEMENTS, A. T., MAO, Y., PESTEREV, A., KAASHOEK, M. F., MORRIS, R., AND ZELDOVICH, N. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10. → pages 50, 52, 53, 75, 76

[22] BRESSOUD, T. C., AND SCHNEIDER, F. B. Hypervisor-based Fault Tolerance. In *Proceedings of the 15th ACM SIGOPS Symposium on Operating Systems Principles* (1995), SOSP'95. → page 7

[23] BRUENING, D., GARNETT, T., AND AMARASINGHE, S. An Infrastructure for Adaptive Dynamic Optimization. In *Proceedings of the International Symposium on Code Generation and Optimization* (2003), CGO'03. → pages 63, 64, 79

[24] BRUMLEY, D., AND SONG, D. Privtrans: Automatically Partitioning Programs for Privilege Separation. In *Proceedings of the 13th USENIX Security Symposium* (2004), Security'04. → page 44

[25] BURROWS, M., ERLINGSSON, U., LEUNG, S.-T. A., VANDEVOORDE, M. T., WALDSPURGER, C. A., WALKER, K., AND WEIHL, W. E. Efficient and Flexible Value Sampling. In *Proceedings of the 9th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems (2000), ASPLOS'00. → page 58

[26] CALDER, B., WANG, J., OGUS, A., NILAKANTAN, N., SKJOLSVOLD, A., MCKELVIE, S., XU, Y., SRIVASTAV, S., WU, J., SIMITCI, H., HARIDAS, J., UDDARAJU, C., KHATRI, H., EDWARDS, A., BEDEKAR, V., MAINALI, S., ABBASI, R., AGARWAL, A., HAQ, M. F. U., HAQ, M. I. U., BHARDWAJ, D., DAYANAND, S., ADUSUMILLI, A., MCNETT, M., SANKARAN, S., MANIVANNAN, K., AND RIGAS, L. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles* (2011), SOSP'11. → page 83

[27] CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. Microreboot — A Technique for Cheap Recovery. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2004), OSDI'04. → pages 10, 22

[28] CANTRILL, B., AND BONWICK, J. Real-World Concurrency. *Queue 6*, 5 (Sept. 2008). → page 88

[29] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing Safe, User Space Access to Fast, Solid State Disks. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12. → page 83

[30] CAULFIELD, A. M., AND SWANSON, S. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (2013), ISCA'13. → pages 6, 82, 83

[31] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2008), ASPLOS'08. → page 43

[32] CITRIX. Citrix XenServer 5.6 Admininistrator's Guide, June 2010. → page 37

[33] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., AND WARFIELD, A. Live Migration of Virtual Machines. In *Proceedings of the 2nd USENIX Conference on Networked Systems Design and Implementation* (2005), NSDI'05. → pages 7, 43, 47

[34] COLP, P. [Xen-devel] [ANNOUNCE] xen ocaml tools. `http://lists.xensource.com/archives/html/xen-devel/2009-02/msg00229.html`, 2009. → page 15

[35] COLP, P., NANAVATI, M., ZHU, J., AIELLO, W., COKER, G., DEEGAN, T., LOSCOCCO, P., AND WARFIELD, A. Breaking Up is Hard to Do: Security and Functionality in a Commodity Hypervisor. In *Proceedings of the 23rd ACM SIGOPS Symposium on Operating Systems Principles* (2011), SOSP'11. → pages v, 2

[36] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O Through Byte-addressable, Persistent Memory. In *Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), SOSP'09. → page 86

[37] CONFLUENT. Kafka Production Deployment. `http://docs.confluent.io/2.0.1/kafka/deployment.html`, 2015. → pages 5, 85

[38] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., HUTCHINSON, N., AND WARFIELD, A. Remus: High Availability via Asynchronous Virtual Machine Replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), NSDI'08. → page 7

[39] CULLY, B., WIRES, J., MEYER, D., JAMIESON, K., FRASER, K., DEEGAN, T., STODDEN, D., LEFEBVRE, G., FERSTAY, D., AND WARFIELD, A. Strata: Scalable High-performance Storage on Virtualized Non-volatile Memory. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), FAST'14. → pages 87, 112

[40] DAWES, B., ABRAHAMS, D., AND RIVERA, R. Boost C++ libraries. `http://www.boost.org`, 2009. → page 53

[41] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th USENIX*

Conference on Operating Systems Design and Implementation (2004), OSDI'04. → page 117

[42] DICE, D. False sharing induced by card table marking. `https://blogs.oracle.com/dave/entry/false_sharing_induced_by_card`, February 2011. → pages 53, 75, 76

[43] DINABURG, A., ROYAL, P., SHARIF, M., AND LEE, W. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (2008), CCS'08. → page 43

[44] DRAGOJEVIĆ, A., NARAYANAN, D., HODSON, O., AND CASTRO, M. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14. → page 104

[45] EIZENBERG, A., HU, S., POKAM, G., AND DEVIETTI, J. Remix: Online Detection and Repair of Cache Contention for the JVM. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2016), PLDI'16. → page 5

[46] EMC. DSSD D5. `https://www.emc.com/en-us/storage/flash/dssd/dssd-d5/index.htm`, 2016. → pages 6, 82

[47] ERLINGSSON, U., ABADI, M., VRABLE, M., BUDIU, M., AND NECULA, G. C. XFI: Software Guards for System Address Spaces. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation* (2006), OSDI'06. → page 65

[48] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12. → page 4

[49] FORD, B., AND COX, R. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the 2008 USENIX Annual Technical Conference* (2008), ATC'08. → page 69

[50] FORD, D., LABELLE, F., POPOVICI, F. I., STOKELY, M., TRUONG, V.-A., BARROSO, L., GRIMES, C., AND QUINLAN, S. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10. → page 83

[51] FRASER, K., HAND, S., NEUGEBAUER, R., PRATT, I., WARFIELD, A., AND WILLIAMSON, M. Safe Hardware Access with the Xen Virtual Machine Monitor. In *Proceedings of the 1st Workshop on Operating System and Architectural Support for Infrastructure* (2004), OASIS'04. → pages 14, 43

[52] GANGER, G. R., MCKUSICK, M. K., SOULES, C. A. N., AND PATT, Y. N. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Trans. Comput. Syst. 18*, 2 (May 2000). → page 91

[53] GIBSON, G. A., NAGLE, D. F., AMIRI, K., BUTLER, J., CHANG, F. W., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. A Cost-effective, High-bandwidth Storage Architecture. In *Proceedings of the 8th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (1998), ASPLOS'98. → page 112

[54] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. The Taser Intrusion Recovery System. In *Proceedings of the 20th ACM SIGOPS Symposium on Operating Systems Principles* (2005), SOSP'05. → page 21

[55] GREATHOUSE, J. L., MA, Z., FRANK, M. I., PERI, R., AND AUSTIN, T. Demand-driven Software Race Detection Using Hardware Performance Counters. In *Proceedings of the 38th Annual International Symposium on Computer Architecture* (2011), ISCA'11. → page 79

[56] GREATHOUSE, J. L., XIN, H., LUO, Y., AND AUSTIN, T. A Case for Unlimited Watchpoints. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12. → page 80

[57] GULATI, A., AHMAD, I., AND WALDSPURGER, C. A. PARDA: Proportional Allocation of Resources for Distributed Storage Access.

In *Proccedings of the 7th USENIX Conference on File and Storage Technologies* (2009), FAST'09. → page 96

[58] GULATI, A., MERCHANT, A., AND VARMAN, P. J. pClock: An Arrival Curve Based Approach for QoS Guarantees in Shared Storage Systems. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (2007), SIGMETRICS'07. → page 97

[59] GULATI, A., MERCHANT, A., AND VARMAN, P. J. mClock: Handling Throughput Variability for Hypervisor IO Scheduling. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (2010), OSDI'10. → page 96

[60] GÜNTHER, S. M., AND WEIDENDORFER, J. Assessing Cache False Sharing Effects by Dynamic Binary Instrumentation. In *Proceedings of the Workshop on Binary Instrumentation and Applications* (2009), WBIA'09. → pages 58, 79

[61] GUO, C., WU, H., DENG, Z., SONI, G., YE, J., PADHYE, J., AND LIPSHTEYN, M. RDMA over Commodity Ethernet at Scale. In *Proceedings of the 2016 ACM SIGCOMM Conference* (2016), SIGCOMM'16. → pages 104, 105

[62] GUPTA, D., LEE, S., VRABLE, M., SAVAGE, S., SNOEREN, A. C., VARGHESE, G., VOELKER, G. M., AND VAHDAT, A. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (2008), OSDI'08. → page 43

[63] HAO, M., SOUNDARARAJAN, G., KENCHAMMANA-HOSEKOTE, D., CHIEN, A. A., AND GUNAWI, H. S. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies* (2016), FAST'16. → page 98

[64] HARDY, N. KeyKOS Architecture. *SIGOPS Oper. Syst. Rev. 19*, 4 (Oct. 1985). → page 42

[65] HENNESSY, J. L., AND PATTERSON, D. A. *Computer Architecture: A Quantitative Approach*, 5 ed. Morgan Kaufmann, 2011. → page 50

[66] HERLIHY, M., AND MOSS, J. E. B. System for achieving atomic non-sequential multi-word operations in shared memory, June 1995. US Patent 5,428,761. → page 53

[67] HITZ, D., LAU, J., AND MALCOLM, M. File System Design for an NFS File Server Appliance. In *Proceedings of the 1994 USENIX Winter Technical Conference* (1994). → page 86

[68] HOEFLER, T., ROSS, R. B., AND ROSCOE, T. Distributing the Data Plane for Remote Storage Access. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems* (2015), HotOS'15. → pages 6, 83

[69] HOHMUTH, M., PETER, M., HÄRTIG, H., AND SHAPIRO, J. S. Reducing TCB Size by Using Untrusted Components: Small Kernels Versus Virtual-machine Monitors. In *Proceedings of the 11th ACM SIGOPS European Workshop* (2004), EW'11. → page 11

[70] HOWARD, J., AND ET AL. A 48-Core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *Proceedings of the IEEE International Solid-State Circuits Conference* (2010), ISSCC'10. → page 50

[71] HUANG, C., SIMITCI, H., XU, Y., OGUS, A., CALDER, B., GOPALAN, P., LI, J., AND YEKHANIN, S. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), ATC'12. → page 83

[72] HUANG, J., BADAM, A., CAULFIELD, L., NATH, S., SENGUPTA, S., SHARMA, B., AND QURESHI, M. K. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17. → page 113

[73] INTEL. Avoiding and Identifying False Sharing Among Threads. `http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/`, November 2011. → page 59

[74] INTEL. Data Plane Development Kit. `http://dpdk.org`, 2012. → page 88

[75] INTEL. Intel Performance Tuning Utility. `http://software.intel.com/en-us/articles/intel-performance-tuning-utility/`, October 2012. → page 79

[76] INTEL. Intel Solid State Drive DC P3700 Series. http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/ssd-dc-p3700-spec.pdf, October 2015. → page 105

[77] INTEL. Storage Plane Development Kit. https://01.org/spdk, 2015. → page 88

[78] ISLAM, N. S., RAHMAN, M. W., JOSE, J., RAJACHANDRASEKAR, R., WANG, H., SUBRAMONI, H., MURTHY, C., AND PANDA, D. K. High Performance RDMA-based Design of HDFS over InfiniBand. In *Proceedings of the IEEE International Conference on High Performance Computing, Networking, Storage and Analysis* (2012), SC'12. → page 104

[79] JALEEL, A., COHN, R. S., KEUNG LUK, C., AND JACOB, B. CMP$im: A Pin-Based On-The-Fly Multi-Core Cache Simulator. In *Proceedings of the 4th Annual Workshop on Modeling, Benchmarking and Simulation* (2008), MOBS'08. → pages 58, 79

[80] JEONG, E. Y., WOO, S., JAMSHED, M., JEONG, H., IHM, S., HAN, D., AND PARK, K. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14. → pages 102, 103

[81] KALIA, A., KAMINSKY, M., AND ANDERSEN, D. G. Using RDMA Efficiently for Key-value Services. In *Proceedings of the 2014 ACM SIGCOMM Conference* (2014), SIGCOMM'14. → page 104

[82] KANEV, S., DARAGO, J. P., HAZELWOOD, K., RANGANATHAN, P., MOSELEY, T., WEI, G.-Y., AND BROOKS, D. Profiling a Warehouse-scale Computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), ISCA'15. → page 4

[83] KAPOOR, R., PORTER, G., TEWARI, M., VOELKER, G. M., AND VAHDAT, A. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012), SoCC'12. → page 112

[84] KAPPEL, K., VELTE, A., AND VELTE, T. *Microsoft Virtualization with Hyper-V*, 1st ed. McGraw-Hill, 2010. → page 8

[85] KARLSSON, M., KARAMANOLIS, C., AND ZHU, X. Triage: Performance Differentiation for Storage Systems Using Adaptive Control. *ACM Transactions on Storage (TOS) 1*, 4 (Nov. 2005). → pages 96, 97

[86] KATZ, R. H. High Performance Network and Channel-Based Storage. Tech. Rep. UCB/CSD-91-650, EECS Department, University of California, Berkeley, Sep 1991. → page 111

[87] KELLER, E., SZEFER, J., REXFORD, J., AND LEE, R. B. NoHype: Virtualized Cloud Infrastructure Without the Virtualization. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (2010), ISCA'10. → page 43

[88] KILPATRICK, D. Privman: A Library for Partitioning Applications. In *Proceedings of the 2003 USENIX Annual Technical Conference* (2003), ATC'03. → page 44

[89] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. `kvm`: the Linux Virtual Machine Monitor. In *Proceedings of the Linux Symposium* (2007). → page 44

[90] KLEIN, G., ELPHINSTONE, K., HEISER, G., ANDRONICK, J., COCK, D., DERRIN, P., ELKADUWE, D., ENGELHARDT, K., KOLANSKI, R., NORRISH, M., SEWELL, T., TUCH, H., AND WINWOOD, S. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (2009), SOSP'09. → page 44

[91] KLIMOVIC, A., KOZYRAKIS, C., THERESKA, E., JOHN, B., AND KUMAR, S. Flash storage disaggregation. In *Proceedings of the 11th ACM SIGOPS European Conference on Computer Systems* (2016), EuroSys'16. → pages 82, 83, 84

[92] KROAH-HARTMAN, G. `udev`: A Userspace Implementation of devfs. In *Proceedings of the Linux Symposium* (2003). → page 17

[93] KUNZ, G. Impact of Shared Storage on Apache Cassandra. `http://www.datastax.com/dev/blog/impact-of-shared-storage-on-apache-cassandra`, January 2017. → pages 5, 85

[94] KUTCH, P. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology. `http://download.intel.com/design/`

network/applnots/321211.pdf, January 2011. → pages 14, 30

[95] LE, M., AND TAMIR, Y. ReHype: Enabling VM Survival Across Hypervisor Failures. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2011), VEE'11. → page 46

[96] LEE, E. K., AND THEKKATH, C. A. Petal: Distributed Virtual Disks. In *Proceedings of the 7th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (1996), ASPLOS'96. → page 112

[97] LEE, S., LIU, M., JUN, S., XU, S., KIM, J., AND ARVIND, A. Application-managed Flash. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (2016), FAST'16, pp. 339–353. → page 113

[98] LEVINTHAL, D. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 processors. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf, 2008. → page 50

[99] LIM, H., HAN, D., ANDERSEN, D. G., AND KAMINSKY, M. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (2014), NSDI'14. → page 112

[100] LITTY, L., LAGAR-CAVILLA, H. A., AND LIE, D. Hypervisor Support for Identifying Covertly Executing Binaries. In *Proceedings of the 17th USENIX Security Symposium* (2008), Security'08. → page 43

[101] LIU, T., AND BERGER, E. D. SHERIFF: Precise Detection and Automatic Mitigation of False Sharing. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications* (2011), OOPSLA'11. → pages 51, 60, 77, 78

[102] LOSCOCCO, P., AND SMALLEY, S. Integrating Flexible Support for Security Policies into the Linux Operating System. In *Proceedings of the 2001 USENIX Annual Technical Conference* (ATC'01). → page 43

[103] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (2005), PLDI'05. → pages 63, 70, 79

[104] LUMB, C. R., MERCHANT, A., AND ALVAREZ, G. A. Facade: Virtual Storage Devices with Performance Guarantees. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (2003), FAST'03. → page 96

[105] MARINOS, I., WATSON, R. N., AND HANDLEY, M. Network Stack Specialization for Performance. In *Proceedings of the 2014 ACM SIGCOMM Conference* (2014), SIGCOMM'14. → page 103

[106] MARTIN, M. M. K., HILL, M. D., AND SORIN, D. J. Why On-chip Cache Coherence is Here to Stay. *Communications of the ACM 55*, 7 (July 2012). → page 50

[107] MCMCC. false sharing in boost::detail::spinlock_pool? `http://stackoverflow.com/questions/11037655/false-sharing-in-boostdetailspinlock-pool`, June 2012. → pages 53, 75, 76

[108] MILLER, E. L., LONG, D. D. E., FREEMAN, W. E., AND REED, B. Strong Security for Network-Attached Storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies* (2002), FAST'02. → page 112

[109] MILLER, J. Why does my choice of storage matter with Cassandra? `http://www.slideshare.net/johnny15676/why-does-my-choiceofstorage-matterwithcassandra`, November 2014. → pages 5, 85

[110] MIŁÓS, G., MURRAY, D. G., HAND, S., AND FETTERMAN, M. A. Satori: Enlightened Page Sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference* (2009), ATC'09. → page 43

[111] MITCHELL, C., GENG, Y., AND LI, J. Using One-sided RDMA Reads to Build a Fast, CPU-efficient Key-value Store. In *Proceedings of the 2013 USENIX Annual Technical Conference* (2013), ATC'13. → page 104

[112] MOLKA, D., HACKENBERG, D., SCHONE, R., AND MULLER, M. S. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques* (2009), PACT'09. → page 51

[113] MOORE, K. E., BOBBA, J., MORAVAN, M. J., HILL, M. D., AND WOOD, D. A. LogTM: Log-based transactional memory. In *Proceedings of the 12th IEEE International Symposium on High Performance Computer Architecture* (2006), HPCA'06. → page 53

[114] MOORE, R. J. A Universal Dynamic Trace for Linux and Other Operating Systems. In *Proceedings of the 2001 USENIX Annual Technical Conference* (2001), ATC'01. → page 63

[115] MORGAN, T. P. Intel Shows Off 3D XPoint Memory Performance. `http://www.nextplatform.com/2015/10/28/intel-shows-off-3d-xpoint-memory-performance/`, October 2015. → page 109

[116] MURRAY, D. G., MILOS, G., AND HAND, S. Improving Xen Security Through Disaggregation. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2008), VEE'08. → pages 11, 43

[117] NANAVATI, M., SPEAR, M., TAYLOR, N., RAJAGOPALAN, S., MEYER, D. T., AIELLO, W., AND WARFIELD, A. Whose Cache Line is it Anyway? Operating System Support for Live Detection and Repair of False Sharing. In *Proceedings of the 8th ACM SIGOPS European Conference on Computer Systems* (2013), EuroSys'13. → pages v, 2

[118] NANAVATI, M., WIRES, J., AND WARFIELD, A. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation* (2017), NSDI'17. → pages v, 2

[119] OLSZEWSKI, M., MIERLE, K., CZAJKOWSKI, A., AND BROWN, A. D. JIT Instrumentation: A Novel Approach to Dynamically Instrument Operating Systems. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007* (2007), EuroSys '07. → pages 63, 65

[120] OLSZEWSKI, M., ZHAO, Q., KOH, D., ANSEL, J., AND AMARASINGHE, S. Aikido: Accelerating Shared Data Dynamic Analyses. In *Proceedings of the 17th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2012), ASPLOS'12. → page 79

[121] OUYANG, J., LIN, S., JIANG, S., HOU, Z., WANG, Y., AND WANG, Y. SDF: Software-defined Flash for Web-scale Internet Storage Systems. In *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2014), ASPLOS'14. → page 83

[122] PAPAMARCOS, M. S., AND PATEL, J. H. A Low-overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th Annual International Symposium on Computer Architecture* (1984), ISCA'84. → page 50

[123] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data* (1988), SIGMOD'88. → page 83

[124] PESTEREV, A., ZELDOVICH, N., AND MORRIS, R. T. Locating Cache Performance Bottlenecks Using Data Profiling. In *Proceedings of the 5th ACM European Conference on Computer Systems* (2010), EuroSys'10. → page 79

[125] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (2014), OSDI'14. → page 103

[126] PETERSEN, C. Introducing Lightning: A flexible NVMe JBOF. `https://code.facebook.com/posts/989638804458007/introducing-lightning-a-flexible-nvme-jbof/`, March 2016. → page 82

[127] PHANISHAYEE, A., KREVAT, E., VASUDEVAN, V., ANDERSEN, D. G., GANGER, G. R., GIBSON, G. A., AND SESHAN, S. Measurement and Analysis of TCP Throughput Collapse in Cluster-based Storage Systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (2008), FAST'08. → page 100

[128] PROVOS, N., FRIEDL, M., AND HONEYMAN, P. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003), Security'03. → page 44

[129] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating MapReduce for Multi-core and Multiprocessor Systems. In *Proceedings of the 13th IEEE International Symposium on High Performance Computer Architecture* (2007), HPCA'07. → pages 51, 75, 77

[130] RILEY, D. Intel SSD DC P3700 800GB and 1.6TB Review: The Future of Storage. `http://www.tomshardware.com/reviews/intel-ssd-dc-p3700-nvme,3858-5.html`, August 2014. → page 106

[131] RIZZO, L. Netmap: A Novel Framework for Fast Packet I/O. In *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), ATC'12. → page 103

[132] RUTKOWSKA, J., AND WOJTCZUK, R. Qubes OS Architecture. `http://qubes-os.org/`, January 2010. → page 43

[133] SAILER, R., JAEGER, T., VALDEZ, E., CACERES, R., PEREZ, R., BERGER, S., GRIFFIN, J. L., AND DOORN, L. V. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference* (2005), ACSAC'05. → page 43

[134] SCHROEDER, B., LAGISETTY, R., AND MERCHANT, A. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies* (2016), FAST'16. → page 94

[135] SCOTT, D., SHARP, R., GAZAGNAIRE, T., AND MADHAVAPEDDY, A. Using Functional Programming Within an Industrial Product Group: Perspectives and Perceptions. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming* (2010), ICFP'10. → page 43

[136] SESHADRI, A., LUK, M., QU, N., AND PERRIG, A. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles* (2007), SOSP'07. → pages 9, 42

[137] SHAPIRO, J. S., SMITH, J. M., AND FARBER, D. J. EROS: A Fast Capability System. In *Proceedings of the 17th ACM SIGOPS Symposium on Operating Systems Principles* (1999), SOSP'99. → page 42

[138] SHEN, Z., CHEN, F., JIA, Y., AND SHAO, Z. DIDACache: A Deep Integration of Device and Application for Flash-Based Key-Value Caching. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17. → page 113

[139] SHINAGAWA, T., EIRAKU, H., TANIMOTO, K., OMOTE, K., HASEGAWA, S., HORIE, T., HIRANO, M., KOURAI, K., OYAMA, Y., KAWAI, E., KONO, K., CHIBA, S., SHINJO, Y., AND KATO, K. BitVisor: A Thin Hypervisor for Enforcing I/O Device Security. In *Proceedings of the 5th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2009), VEE'09. → pages 9, 42

[140] SHREEDHAR, M., AND VARGHESE, G. Efficient Fair Queueing Using Deficit Round Robin. In *Proceedings of the 1995 ACM SIGCOMM Conference* (1995), SIGCOMM'95. → page 100

[141] SHUE, D., AND FREEDMAN, M. J. From Application Requests to Virtual IOPs: Provisioned Key-value Storage with Libra. In *Proceedings of the 9th ACM SIGOPS European Conference on Computer Systems* (2014), EuroSys'14. → pages 96, 99

[142] SHUE, D., FREEDMAN, M. J., AND SHAIKH, A. Performance Isolation and Fairness for Multi-tenant Cloud Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (2012), OSDI'12. → page 96

[143] SPANJER, E., AND HO, E. The Why and How of SSD Performance Benchmarking. `http://www.snia.org/sites/default/education/tutorials/2011/fall/SolidState/EstherSpanjer_The_Why_How_SSD_Performance_Benchmarking.pdf`, 2011. → page 106

[144] SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSEN, D., AND LEPREAU, J. The Flask Security Architecture: System Support for Diverse Security Policies. In *Proceedings of the 8th USENIX Security Symposium* (1999), Security'99. → page 20

[145] STEINBERG, U., AND KAUER, B. NOVA: A Microhypervisor-based Secure Virtualization Architecture. In *Proceedings of the 5th European Conference on Computer Systems* (2010), EuroSys'10, ACM. → pages 9, 43

[146] STONEBRAKER, M., MADDEN, S., ABADI, D. J., HARIZOPOULOS, S., HACHEM, N., AND HELLAND, P. The End of an Architectural Era: (It's Time for a Complete Rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases* (2007), VLDB'07. → page 112

[147] SWANSON, S., AND CAULFIELD, A. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *Computer 46*, 8 (Aug. 2013). → page 82

[148] TAMCHES, A., AND MILLER, B. P. Fine-grained Dynamic Instrumentation of Commodity Operating System Kernels. In *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation* (1999), OSDI'99. → pages 63, 64

[149] TANENBAUM, A. S., HERDER, J. N., AND BOS, H. Can We Make Operating Systems Reliable and Secure? *IEEE Computer 39*, 5 (May 2006). → page 9

[150] TAO, J., AND KARL, W. CacheIn: A Toolset for Comprehensive Cache Inspection. In *Proceedings of the 5th International Conference on Computational Science* (2005), ICCS'05. → page 79

[151] THACKER, C. Beehive: A many-core computer for FPGAs (v5). `http://projects.csail.mit.edu/beehive/BeehiveV5.pdf`, Jan 2010. → page 50

[152] THEKKATH, C. A., MANN, T., AND LEE, E. K. Frangipani: A Scalable Distributed File System. In *Proceedings of the 16th ACM SIGOPS Symposium on Operating Systems Principles* (1997), SOSP'97. → pages 87, 112

[153] THERESKA, E., BALLANI, H., O'SHEA, G., KARAGIANNIS, T., ROWSTRON, A., TALPEY, T., BLACK, R., AND ZHU, T. IOFlow: A Software-defined Storage Architecture. In *Proceedings of the 24th ACM SIGOPS Symposium on Operating Systems Principles* (2013), SOSP'13. → page 96

[154] THIBAULT, S., AND DEEGAN, T. Improving Performance by Embedding HPC Applications in Lightweight Xen Domains. In *Proceedings of the 2nd Workshop on System-level Virtualization for High Performance Computing* (2008), HPCVirt'08. → pages 14, 44

[155] TSIROGIANNIS, D., HARIZOPOULOS, S., AND SHAH, M. A. Analyzing the Energy Efficiency of a Database Server. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2010), SIGMOD'10. → page 7

[156] VMWARE. VMware vSphere Storage APIs âĂŞ Array Integration (VAAI). `http://www.vmware.com/resources/techresources/10337`, December 2012. → pages 86, 94

[157] WACHS, M., ABD-EL-MALEK, M., THERESKA, E., AND GANGER, G. R. Argon: Performance Insulation for Shared Storage Servers. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies* (2007), FAST'07. → page 96

[158] WANG, A., VENKATARAMAN, S., ALSPAUGH, S., KATZ, R., AND STOICA, I. Cake: Enabling High-level SLOs on Shared Storage Systems. In *Proceedings of the 3rd ACM Symposium on Cloud Computing* (2012), SoCC'12. → pages 96, 97

[159] WANG, Z., JIANG, X., CUI, W., AND NING, P. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (2009), CCS'09. → page 43

[160] WILKES, J., MOGUL, J., AND SUERMONDT, J. Utilification. In *Proceedings of the 11th ACM SIGOPS European Workshop* (2004), EW'11. → page 7

[161] WIRES, J., AND WARFIELD, A. Mirador: An Active Control Plane for Datacenter Storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies* (2017), FAST'17. → page 101

[162] WITCHEL, E., CATES, J., AND ASANOVIĆ, K. Mondrian Memory Protection. In *Proceedings of the 10th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2002), ASPLOS'02. → page 5

[163] WORKGROUP, N. E. NVM Express revision 1.2 Specification. http://nvmexpress.org/wp-content/uploads/NVM_Express_1_2_Gold_20141209.pdf, November 2014. → pages 83, 90

[164] YANG, J., MINTURN, D. B., AND HADY, F. When Poll is Better Than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies* (2012), FAST'12. → page 83

[165] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the 30th IEEE Symposium on Security and Privacy* (2009), S&P'09. → page 69

[166] ZHAO, Q., KOH, D., RAZA, S., BRUENING, D., WONG, W.-F., AND AMARASINGHE, S. Dynamic Cache Contention Detection in Multi-threaded Applications. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2011), VEE'11. → pages 51, 58, 79

[167] ZHU, T., TUMANOV, A., KOZUCH, M. A., HARCHOL-BALTER, M., AND GANGER, G. R. PriorityMeister: Tail Latency QoS for Shared Networked Storage. In *Proceedings of the 5th ACM Symposium on Cloud Computing* (2014), SOCC'14. → pages 96, 97

[168] ZHU, Y., ERAN, H., FIRESTONE, D., GUO, C., LIPSHTEYN, M., LIRON, Y., PADHYE, J., RAINDEL, S., YAHIA, M. H., AND ZHANG, M. Congestion Control for Large-Scale RDMA Deployments. In *Proceedings of the 2015 ACM SIGCOMM Conference* (2015), SIGCOMM'15. → page 105