The Serverless Datacenter: Hardware and Software Techniques for Resource Disaggregation

Nathan Pemberton

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2022-86
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-86.html

May 13, 2022

The Serverless Datacenter: Hardware and Software Techniques for Resource Disaggregation

by

Nathan Trawick Pemberton

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Randy Katz, Co-chair
Professor Joseph Gonzalez, Co-chair
Assistant Professor Yakun Sophia Shao
Dr. Kimberly Keeton

Spring 2022

The Serverless Datacenter: Hardware and Software Techniques for Resource Disaggregation

Abstract

The Serverless Datacenter: Hardware and Software Techniques for Resource Disaggregation

by

Nathan Trawick Pemberton

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Randy Katz, Co-chair

Professor Joseph Gonzalez, Co-chair

Datacenters have grown beyond a simple collection of independent computers. They are now a complex and interconnected ecosystem of heterogeneous hardware and software services: a warehouse-scale computer. These computers are wildly expensive to provision and operate, yet we struggle to effectively utilize them. It is not uncommon to have half of allocated resources unused, while other resources cannot be allocated at all. Resource needs vary widely, both between jobs, and even over time within a single job. When we aggregate resources into fixed "slots" (i.e., servers), we take away the flexibility needed to accommodate these varying needs. I propose a different approach: resource disaggregation. Rather than requiring the system to fit jobs into fixed-sized servers, we make any resource in the system available to any job (physical disaggregation). Rather than requiring jobs to allocate all their resources up-front, we allow them to allocate resources only when they actually need them (logical disaggregation). I argue that unlocking the full potential of physical disaggregation requires moving logical interfaces to a fundamentally disaggregated paradigm. Likewise, logically disaggregated systems can provide some benefit on today's hardware, but only reach their full potential when co-designed with physically disaggregated hardware. In this dissertation, I present tools and methodologies I developed to support that hardware/software co-design. I then describe how I used these tools to implement a simple hardware accelerator that works with the operating system to improve the performance of physically disaggregated memory. I evaluate it with end-to-end benchmarks in RTL simulation and find that it reduces the latency of remote memory access by 2.2x and improves end-to-end performance by 20 % over a software-only approach. Next, I show how I extended the logically disaggregated serverless programming model to heterogeneous compute resources. My prototype achieves 50x better performance with fewer resources than today's aggregated approaches. Together, these techniques form a vision of a serverless datacenter that unlocks the promise of pay-per-use and rapid innovation that warehouse-scale computers should provide.

Dedicated to:
The ideals of compassion, understanding, and constant personal growth.

"I notice that if you have the door to your office closed, you get more work done today and tomorrow, and you are more productive than most. But 10 years later somehow you don't quite know what problems are worth working on" - Richard W. Hamming

Contents

Acknowledgments

They say it takes a village to raise a child. I believe that holds for doctors of philosophy as well. So many have helped me along the way that I could not possibly thank them all. That being said, there are a few that call for special mention. First, I'd like to thank the teachers that encouraged and advocated for me through every level of education: Donald Repucci (Morro Bay High), Randy Skovil (Cuesta Community College), and Ethan Miller (UC Santa Cruz). Next, the professors that supported me at Berkeley: John Kubiatowicz, Randy Katz, Krste Asanović, Ion Stoica, Joe Hellerstein, Sophia Shao, and Joey Gonzalez. I especially want to thank my mentor and unofficial advisor, Kim Keeton, who has been a constant source of support and advice. My friends and colleagues are too numerous to list, but you know who you are. I must also thank my siblings, aunts, uncles, and grandparents who have always been there for me. Finally, if anyone deserves credit for getting me to where I am, it's my parents Joel and Teresa Pemberton. Thank you for teaching me curiosity, compassion, and communication. Even though it's a cliché, it's no less true: I couldn't have done it without you.

Chapter 1

Introduction

Warehouse-scale computers (WSCs), and the cloud in particular, present a unique set of goals and challenges when compared to traditional single-node systems. Unlike PCs or individual servers, warehouse-scale computers are *multitenant* and can dynamically allocate resources to different users. In the cloud, these allocations correspond directly to monetary cost, both for the user and for the provider. Since allocations are expensive, a core objective in WSCs is to maximize resource utilization. For the customer, this means that they do not pay for resources that they don't use. For the provider, this means that they can defer capital expenditures by supporting more customer workloads on existing resources. In practice, achieving high utilization is easier said than done. A workload trace released by Google in 2019 showed that average CPU and memory utilization rarely exceeded 60%. This is despite allocating 150% of total resources to account for poor intra-task utilization [267]. This is only slightly better than the 50% utilization reported nearly 10 years earlier [224]. Microsoft similarly reports only 52% average utilization of their deep learning GPUs while Alibaba sees about 50% CPU utilization [123, 106]. Supercomputers also struggle to utilize their resources [282, 66, 176].

At a high level, there are two primary categories of underutilization in WSCs:

- **Stranded Resources:** There are enough resources cluster-wide for a job, but no one server has enough of each resource type (Figure 1.1a). Think of this like external fragmentation [219].

- **Idle Resources:** We have allocated resources to a job, but it doesn't use every resource at all times (Figure 1.3a). Think of this like internal fragmentation [219].

In both cases, the problem stems from a requirement to allocate multiple resources simultaneously. In the case of stranded resources, this is because we have created discrete physical servers that aggregate a relatively small amount of each resource type. For idle resources, we are aggregating multiple resources into a single logical allocation even when the job only needs a subset of them at any given time. As we introduce new resource types like non-volatile memories and accelerators, this problem only gets worse. I propose removing these restrictions by *disaggregating* resources, both physically (Figure 1.1b) and logically (Figure 1.3b). This allows the system to access physical resources from anywhere in the datacenter and the applications to allocate and release individual resources dynamically. While this improves utilization, it often comes at the cost of performance. In this dissertation, I present software and hardware techniques for disaggregation that provide both high performance and high utilization. I also describe the tools and methodologies for hardware/software codesign that I developed to support my research. I implement prototypes of these techniques and evaluate them with end-to-end benchmarks on realistic platforms. For the hardware techniques, this includes synthesizable designs evaluated with cycle-exact simulation. These results suggest that focusing on only one aspect of disaggregation will not be sufficient to reach its full potential. Future systems will need to be disaggregated both *logically* and *physically* to achieve high performance and high utilization in a way that is intuitive to users, and practical for providers.

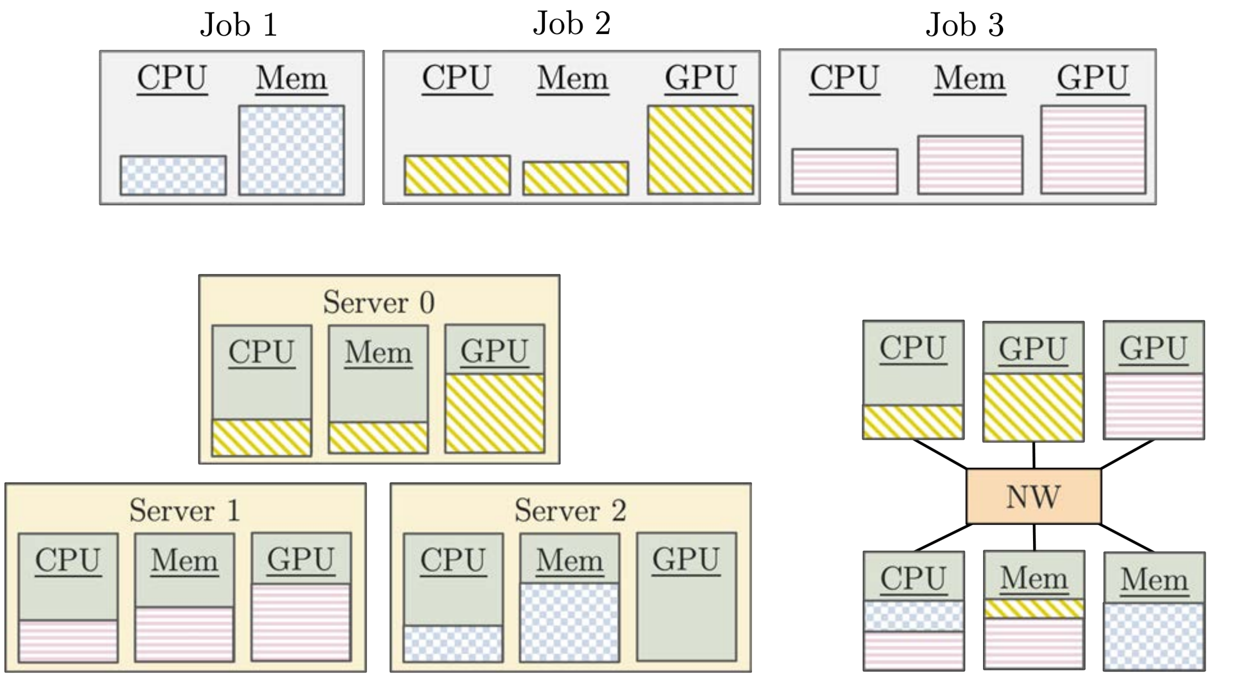
1.1 Physical Disaggregation

Traditional datacenter designs aggregate all necessary resources into many self contained server chassis. This design was motivated by the ability to leverage commodity PC components and networks [12, 260, 33]. Additionally, an aggregated design was desirable because in-chassis interconnects were significantly faster than networks. However, networking technology has seen a rapid increase in performance, with 40 Gbit/s Ethernet becoming commonplace, and 100 Gbit/s networks readily available, narrowing the bandwidth and latency gap between local and remote resources [40]. Workloads have also changed; applications are fundamentally distributed, use larger and rapidly changing datasets, and demand latencies that can only be delivered by in-memory processing [136].

These hardware and software trends have led to proposals from both academia [17, 135] and industry [114, 118, 120, 85] for a new style of WSC where resources are disaggregated. At Berkeley, Krste Asanović proposed such a system, called FireBox, that has served as a conceptual framework for my work (see Figure 1.2) [17]. In a disaggregated WSC, resources like disk and memory become first-class citizens over a high-performance network. This allows datacenter operators to scale resource capacity beyond what fits in a single chassis while allocating it more flexibly [157].

Disaggregated resources are not without their drawbacks. No matter how fast networks become, accessing remote resources always carries a latency penalty. Fault tolerance and allocation policies also become more complex. Reaping the benefits of disaggregation while enabling high performance and reliable applications will require new approaches to both hardware and software.

In this dissertation, I focus on one resource type for physical disaggregation: memory. Disaggregation is particularly important for memory because state, unlike compute, is non-fungible. While any CPU or GPU can run any program right away, state must be explicitly moved in order to free or re-allocate memory. Furthermore, memory demands have exploded



(a) **Physically Aggregated Resources** Servers each contain a relatively small amount of each resource type. Even though Server 2 has enough CPU resources to accommodate both jobs 1 and 2, it has insufficient memory, wasting both a GPU and half of its CPUs.

(b) **Physically Disaggregated Resources** Resources can be accessed from anywhere in the cluster independently. This allows the jobs to pack more densely into the available resources, reducing stranding.

Figure 1.1: **Stranded Resources and Physical Disaggregation** Resources can be stranded when they are physically aggregated into servers. By physically disaggregating resources, we can allocate resources independently and accommodate all jobs more densely.

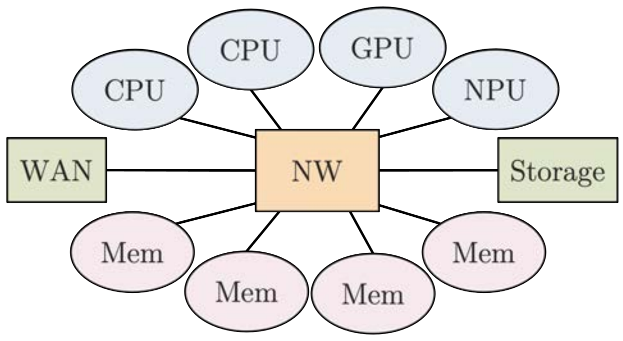
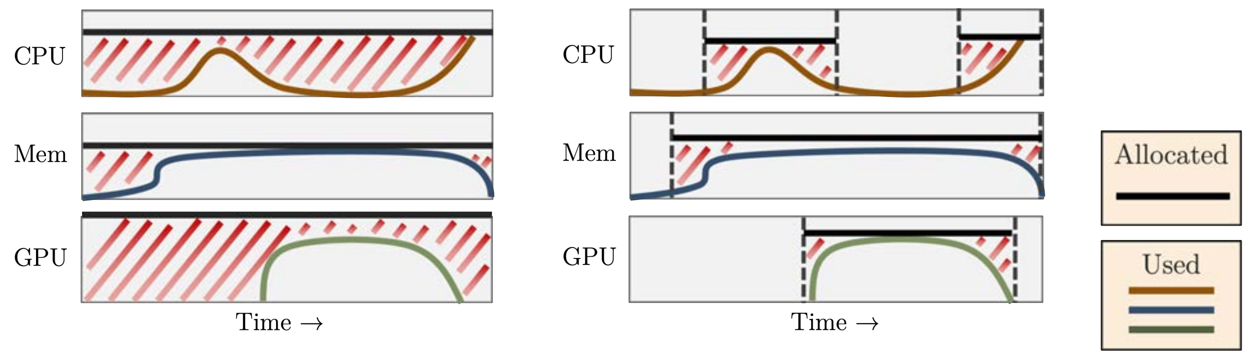


Figure 1.2: Overview of the proposed FireBox warehouse-scale computer



(a) **Logically Aggregated Resources:** Every resource must be allocated to a job for its full duration. Jobs are unlikely to need every resource at all times, causing some to go idle.

(b) **Logically Disaggregated Resources:** Resources can be allocated to jobs independently in time. Jobs consume resources only when needed, minimizing idleness.

Figure 1.3: **Idle Resources and Logical Disaggregation** Resources can go idle when they must be allocated together (logical disaggregation). The shaded regions represent periods where a resource is allocated to a job, but is not being used (idle resources). By allocating and deallocating resources independently, logical disaggregation reduces waste.

in the last decade with the rise of big data and analytics, quickly outstripping the capacity of servers, even if any particular job only touches a subset of it at any given time. In Chapter 4, I present my work on hardware acceleration for paging-based approaches to memory disaggregation along with a broader discussion of the design space.

1.2 Logical Disaggregation

While networking technologies and new chip designs enable direct physical access to remote resources, they make no assertions about how users ought to access these resources. For this, we need a *logical* model of disaggregation (Figure 1.3). This interface can be fully transparent like a distributed operating system [48] or very explicit like web services [276], or anywhere in between (see Figure 1.4).

One interface of particular interest is serverless computing [56, 239]. Serverless structures applications around explicit state (*objects*) and transformations over that state (*functions*). Functions can take many forms, but the most general incarnation is called *Function-as-a-Service (FaaS)*. In FaaS, users supply code in a high level language that is then run in a conventional operating system environment, typically a Linux container. Importantly, functions do not maintain any implicit state of their own; computation is *logically disaggregated* from state. This moves us closer to the ideal of Figure 1.3b. Explicit state means that compute resources can be freed as soon as a function completes, reducing idleness. Short-lived

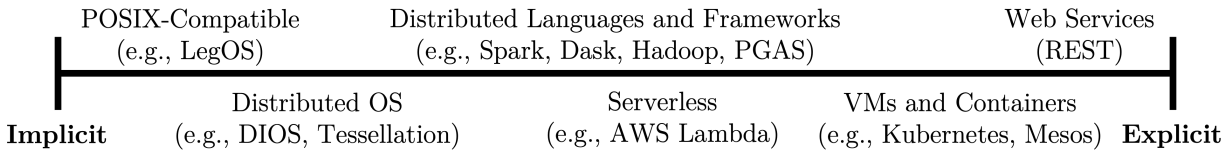


Figure 1.4: Warehouse-scale computer interfaces span a spectrum of very implicit (users don't know it's distributed) to very explicit (users manage all resources and networks).

and fine-grain functions provide more placement flexibility, reducing stranding. By removing the concept of a physical server, serverless computing also greatly simplifies operational concerns like resource provisioning or server maintenance.

In this thesis, I argue that serverless computing provides a good, but incomplete, abstraction for logical disaggregation that maps well to physically disaggregated resources [206, 207]. In Chapter 5, I begin by comparing serverless to the more aggregated interfaces in use today. I then extend the serverless model to other resource types like GPUs. In doing so, I enable systems to effectively utilize these expensive resources, even when they must be shared among many users.

1.3 Hardware/Software Co-Design

Getting the full potential out of disaggregation will require designing hardware and software to work together. This sort of co-design is hard in practice, especially for complex distributed systems running general purpose software. It is not enough to evaluate our hardware with a few bare-metal instructions, nor is it sufficient to run end-to-end software on abstract models of our hardware. Real systems must be designed jointly, informed by constant feedback and rapid iteration.

In the course of my research, I often found that the existing tools and methodologies available to me were insufficient. Simulators were either too slow to be practical, or too abstract to provide useful insights into performance or implementation practicality. Hardware was monolithic and difficult to adapt as end-to-end evaluation identified new problems or requirements. Software workloads were built ad-hoc and were difficult to manage, particularly as hardware designs evolved. These limitations are particularly problematic for research into disaggregation and WSCs. These systems are physically distributed, requiring simulation of multiple networked components. Beyond just the physical, logical disaggregation requires an understanding of full-stack software including operating systems, network protocols, and long-running applications.

To make progress, I needed to design new tools and methodologies for agile hardware development, particularly for software workload management. Simultaneously, others in my research group were developing tools for other aspects of the agile hardware design process. Eventually, we combined these tools into an end-to-end system on chip (SoC) development framework called Chipyard [10]. Chipyard provides tools for low-level VLSI concerns, RTL base designs, simulators and evaluation, and software development. I focus primarily on that last component: software workload management. Software workload management is more than just building the software artifacts like boot-binary and disk image, it is about end-to-end lifecycle management including designing, building, and evaluating.

In Chapter 3, I describe FireMarshal, a tool I built to tackle these challenges. I also go over Chipyard and the hardware/software co-design process more generally. These tools enabled me to quickly build and evaluate realistic systems rather than relying on incomplete or abstract designs.

1.4 What to Expect from this Dissertation

In this dissertation, I describe how to address underutilization in WSCs by disaggregating both our logical programming models, and the hardware those models run on. I do this by presenting several disaggregated systems that I have designed, both logical and physical. While physical disaggregation often hurts performance in favor of utilization, I show that those performance impacts can be partially mitigated through improvements in hardware using one such accelerator as an example. However, the greatest gains came when systems presented a fundamentally disaggregated logical model of computation. These systems could maintain higher performance while using fewer resources than their aggregated counterparts. I argue that moving logical interfaces to a fundamentally disaggregated paradigm is necessary to unlock the potential of physically disaggregated systems.

I begin in Chapter 2 with a look at what warehouse-scale computers are, and how they can be disaggregated using today's state-of-the-art techniques. Chapter 3 lays out the context for the remainder of this dissertation by describing the tools and methodologies I developed to make hardware/software co-design practical and agile. I then use these tools to explore disaggregation from a physical perspective in Chapter 4 where I focus on the topic of memory disaggregation. There, I present an implementation of a hardware accelerator I designed that reduces page fault latency by 2.2x and improves end to end performance by 20 %. I then show how a more explicitly disaggregated interface to process checkpointing sees even greater gains with nearly 4x faster checkpoints than today's less explicit approaches. Next, in Chapter 5, I come from the other direction and explore techniques for logical disaggregation, with a focus on serverless computing and application accelerators. I evaluate a prototype system I developed to present a serverless interface to GPUs that improves throughput by 50x over traditional approaches by more effectively utilizing GPUs. Chapter 6 looks toward the future by proposing ways that logical disaggregation, in the form of serverless computing, can enable a rich vein of new research on cloud system interfaces and hardware. I conclude in Chapter 7 with some reflections on disaggregation, and research practice more broadly. At the end of this document, I provide a list of references and a glossary of terms and abbreviations. In the digital version, references and terminology can be clicked on to link to their definition.

Chapter 2

A Brief History of Warehouse-Scale Computers

In the 1990s, computers were becoming cheaper and smaller while the cost and performance of local networking were rapidly improving. Academics and industry soon began taking advantage of these trends by grouping multiple computers together into tightly coupled clusters [12, 260, 110]. With the emergence of the internet and web, these clusters grew to fill entire buildings; the warehouse-scale computer (WSC) was born [33, 34]. Finally, companies recognized the value in their WSCs and began allowing customers to use their excess capacity (for a fee, of course). This model became known as cloud computing [15]. The cloud model is so lucrative that WSCs are now being designed exclusively to offer cloud services.

WSCs, and the cloud in particular, present a unique set of goals and challenges when compared to traditional single-node systems. Unlike PCs or individual servers, WSCs are *multitenant*. Multitenancy makes security even more critical and challenging while adding the need for performance isolation. On the other hand, multitenancy means that resource demands are averaged over many users, leading to more stable aggregate system load. It also allows providers to dynamically allocate resources to different users as needs arise, though this may make performance less predictable to users. Since clouds are centrally designed and administered, they are incentivized to innovate on both hardware and software systems.

This chapter begins with a brief survey of how providers have specialized their hardware for the WSC setting. In §2.2, I describe the range of system interfaces that are available for WSCs today. I finish in §2.3 with some more recent proposals for a physically disaggregated WSC.

2.1 Hardware Specialization in WSCs

An important consequence of the cloud is the centralization of cost and administration. Traditional vendors of server hardware need to appeal to a broad market and wide range of scales.

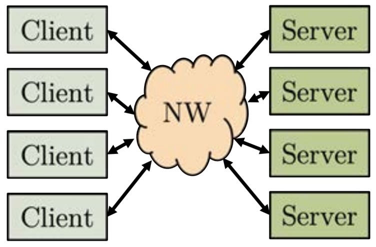
On-premises operators and “server-farm” style datacenters need standardized and modular components. They might buy one server or one hundred servers, and they all need to fit into standard physical slots and network architectures. These market forces work to suppress innovation. Vendors are discouraged from introducing new accelerators because customers would need to make deep changes to their environments to use them. Networks and devices can’t be co-designed, and specialized systems rarely scale beyond a single rack (referred to as *appliances*). WSC operators are different. They manage their entire deployments and invest hundreds of millions of dollars in each one. They compete on raw performance, cost, and features. This centralization and shift in market forces incentivizes innovation. Indeed, cloud and hyperscale operators are investing heavily in custom hardware.

The most high-profile efforts toward custom hardware have come in the form of deep learning and other application accelerators. Google developed a custom system on chip (SoC) for deep learning training in 2015 called the tensor-processing unit (TPU) [127]. These accelerators are deployed in custom clusters called TPU Pods that connect thousands of TPUs with a high performance interconnect [261]. Google also has custom accelerators for video transcoding [220]. Similarly, Amazon built a custom deep learning inference chip called Inferentia that is available directly to users as well as powering their inference services [119]. Microsoft has deployed network-attached FPGAs in their datacenters to support a wide range of tasks, including model serving through their Brainwave project [215, 65]. While Microsoft did not develop a new SoC for this purpose, they worked with Altera (now Intel) to build custom FPGA boards. There are also many startups developing deep learning accelerators that can be deployed in WSCs [155, 5].

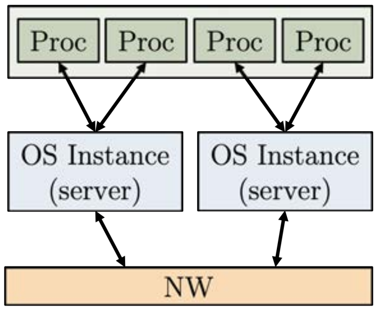
Perhaps less flashy, but equally important, are systems and platform-level accelerators. While storage appliances have been around for a long time, they were typically not customized for particular environments. Now, some hyperscale operators have custom storage hardware systems to optimize density, power, and performance [25, 30]. Amazon built a platform and virtualization chip called Nitro that handles many common cloud-specific functions independently of the target platform, enabling rapid innovation on instance types [156]. Google reported a number of “datacenter taxes”, common tasks that consume significant resources [130]. This has proven a rich vein for research on accelerators for things like memory copying or protocol offloading [170, 133].

2.2 System Interface Specialization in WSCs

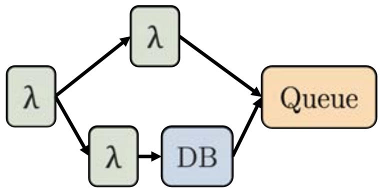
As with hardware, WSCs and the cloud provide an opportunity to re-think our system interfaces. Cloud provider application programming interfaces (APIs) provide resource allocation, protection, communication, naming, and scheduling for their WSCs. These are the same things that a traditional operating system must do. If we are going to think of these systems as warehouse-scale computers, it is worth considering what the operating system interface to this computer ought to be. In other words: What is the POSIX for the cloud? In this section, based on joint work with Johann Schleier-Smith [207], I describe the range



(a) **Web Services:** Applications are structured around static allocations of virtual machines or containers communicating over web protocols like HTTP.



(b) **Distributed Operating System:** The WSC is presented as a single operating system image to users. Applications are structured around conventional processes and files.



(c) **Serverless:** Applications are structured around explicit state, and transformations over that state.

Figure 2.1: System interfaces to WSCs today

of WSC interfaces that exist today.

2.2.1 Virtual Machines, Containers, and Web Services

Arguably the most common interface to the cloud is through servers and web services (Figure 2.1a). Amazon's cloud division is even called "Amazon Web Services". Web services build on the rich ecosystem of internet and web technologies that are known to scale to millions of users. TCP/IP handles routing and flow control at internet scale while HTTP provides a portable and abstract interface to web services. By structuring applications as stateless HTTP endpoints (called REST [88]), we can add load balancers and seamless fail-over mechanisms.

As the cloud emerged, users wanted to continue building applications using this proven architecture. In response, clouds typically provide high-level services through a RESTful API. For user-defined services, they rely on open-ended allocations of virtual machines or containers, mimicking the server farms their customers are used to. A number of systems have arisen to support the provisioning and scheduling of these allocations. Kubernetes (based on Google's internal Borg scheduler [273, 267]) has particularly strong industry adoption [44, 47]. More broadly, there are systems like Mesos or OpenStack that seek to provide a unified view of job scheduling and deployment [199, 300].

While this style of WSC interface does indeed scale, it also brings with it certain disadvantages. Web services assume internet-like latencies between communicating entities. In reality, WSCs support networks with microsecond latencies and hundreds of gigabits per second bandwidths. At these speeds, HTTP processing and redundant computations from stateless APIs become significant overheads [32]. Since servers experience time-varying demands, it can be hard to right-size allocations. This leads to both idle and stranded resources. This problem only gets worse with applications like deep-learning that require expensive specialized hardware. In essence, the web services interface assumes too little about WSC locality, performance, and allocation flexibility.

2.2.2 Distributed Operating Systems

Making a collection of computers work like one powerful computer is a longstanding goal of distributed operating systems research [264, 242]. There was a great deal of research on this topic in the decades after inexpensive workstation hardware and local networks first became available [201, 279, 182, 108, 240, 8, 64, 74, 110, 12]. These efforts generally sought to provide a UNIX-like interface to a group of machines (Figure 2.1b). However, this line of work was largely eclipsed by the emergence of the internet, which ushered in a new era of distributed systems that operated on a far larger scale [33, 34]. The internet technologies won in the market with the help of tremendous investment, which makes it hard to conclude whether POSIX-like distributed operating systems suffered from technical failings, or whether they simply were not ready to meet the needs of gigantic internet services.

In [242], Schwarzkopf, Grosvner, and Hand argue that hardware trends have made warehouse-scale computers suitable for distributed operating systems. Indeed, there have been several recent projects exploring designs in this direction [288, 289, 208, 253, 211, 241, 246, 69].

The problem with POSIX and locality transparent operating system designs is the inverse of the problem with web services. While web services have a built in design assumption that everything is remote, POSIX has the built in assumption that everything is local. NFS provides a clear example of how interfaces designed in a local setting can prove troublesome in a distributed setting. A remote file system that becomes unreachable may cause API responses not possible with a local file system [278]. Compliance with POSIX consistency guarantees [191], notably linearizability [111], has also been a perennial source of pain for distributed file system implementations [187, 281, 113, 102]. This transparency also makes performance less predictable, a significant challenge for large-scale applications. I discuss this effect in more detail in §5.1.4.

Distributed operating systems provide some advantages from the perspective of utilization. They can allocate or re-locate tasks to any idle resources in the system. Since they have greater insight into application resources, they can transparently disaggregate some individual resources [82, 93, 92]. However, this transparent disaggregation introduces performance overheads that applications may not account for. I will give some concrete examples of this effect in §4.2 and §5.1.4. Long-running processes are still assigned resources that are difficult to reclaim dynamically. Likewise, current operating systems allocate accelerators to processes exclusively, whether they use them or not. This resource-centric, locality-unaware, conception of applications fundamentally limits our ability to address sources of underutilization.

2.2.3 Serverless

More recently, we have seen a new programming paradigm emerge in the cloud: Serverless [56]. Serverless computing avoids explicit provisioning of resources in favor of time-bounded invocations of functionality (Figure 2.1c). Users ask for some function or service to be invoked without considering how or where it will run. These functions are typically narrow in purpose and use few resources, favoring multiple invocations rather than a single large function. When the function has completed, the resources are freed. Often, functions are user defined, called *Function-as-a-Service (FaaS)*, though functionality may also take the form of scalable services like a database. Users express applications as a graph of function invocations, often with a common data layer to express state. Since functions have a finite lifetime, they are not permitted to maintain implicit state (i.e., state that is invisible to the provider). Instead, all state must be explicitly persisted to a data layer. This design addresses both sources of underutilization. Since functions are small and fine-grained, many can be packed onto a single server, reducing stranded resources. Since functions always run to completion and lack implicit state, the provider can quickly reclaim idle resources. Serverless is a promising and quickly evolving paradigm in both the academic and industrial communities [239].

Serverless is not without its limitations [109]. I will mention two major drawbacks here. The first is the significant deviation from traditional programming models. Few applications are designed to identify all critical state explicitly and may not be factored into sufficiently fine-grained functionality. While a legitimate concern, cloud users have demonstrated a willingness to adopt new models. Agility is often favored over backwards compatibility. The widely used microservice architecture provides a good example [26]. Microservice architectures break monolithic applications into many independent components communicating through well-defined APIs. This popular technique requires significant re-design of applications but enables organizational flexibility, agility within individual components, and scalability.

Another major drawback of serverless computing is the increased communication and invocation costs of communicating state to many function invocations through a data layer. While a significant concern, it is mitigated somewhat by the observation that logical disaggregation does not imply physical disaggregation (an observation made in [258]). It is often possible to place a subset of functions on the same physical server. More generally, the serverless programming model frees providers to implement novel scheduling, placement, and even hardware techniques [206].

While promising, there remain many questions about how this may be achieved in practice, particularly for new technologies like remote memory or domain-specific accelerators. Despite these challenges, the serverless model is well aligned with the physical realities of modern WSCs. The logically disaggregated state maps to physically disaggregated memory and storage. Functions can be instantiated on any compute resource in the datacenter, regardless of its locality to other resources. In Chapter 5, I argue that these properties make serverless an appealing starting point for a logically disaggregated WSC interface that will enable higher performance and better utilization of physically disaggregated resources.

2.3 The Disaggregated Datacenter

Computer hardware has advanced at an incredible rate since the first internet servers appeared in the 1980s. Networks have gone from a few kbit/s to hundreds of Gbit/s. Proposed integrated silicon photonic networks promise Tbit/s of bandwidth at sub-microsecond latencies [262]. CPUs are many thousands of times faster and their caches have more capacity than the hard drives of a few decades ago. Despite these advances, today's servers follow much the same format as the original PCs that powered the first modern datacenters [260, 12, 33]. These "pizza boxes" contain a power supply, one or more CPUs, local memory, and any number of peripherals. They run a single operating system instance and integrate with the outside world through IP.

The power of these new technologies, especially networking, has motivated a new WSC design: the disaggregated datacenter (DDC) [118, 135, 17]. In a DDC, resources are moved from aggregated multi-resource partitions (servers) into globally accessible resource-specific network endpoints (see Figure 1.2 in Chapter 1). This physical disaggregation minimizes resource stranding by allowing applications to directly access any resource in the system. They also enable flexible system administration and provisioning. Today, WSC operators must decide on the resource mix in their servers up-front. If more of one resource is needed, they must buy more servers, including other resource types that they may not need. If a new resource type emerges, the operator must design and provision a new server type. In a DDC, resources can be scaled independently and new types can be introduced as standalone network endpoints.

While the term "disaggregated datacenter" may imply disaggregation across an entire warehouse, the core techniques can apply at multiple scales. For example, Hewlett-Packard Enterprise sells a rack-scale system with disaggregated memory [179], while Oracle sells a database appliance with globally accessible resources that scales up to 12 racks [3]. DDCs may also focus on a subset of resources rather than disaggregating every resource at once. Today, persistent storage resources are commonly disaggregated [25, 91]. Some operators have disaggregated accelerators for applications like deep learning [261, 215] or data analytics [30].

2.4 Takeaways

Warehouse-scale computers have grown along with the internet and the dramatic expansion of computation across science and industry. While they began as a simple extension of PCs, they have now become highly innovative dedicated systems. The economies of scale and consolidation of system administration in the cloud has driven this innovation to new heights. The highly multitenant nature of these systems also brings the need for high utilization to the forefront. Single-application clusters are sized for peak usage, while average utilization remains a secondary concern. In contrast, modern clouds experience a more consistent average load even though individual tenants remain highly variable. In this new paradigm, average utilization is critical as it brings down costs for users and providers. However, techniques to improve this utilization can hurt performance, particularly in the tail. Providers must carefully manage this trade-off.

Today's systems are far more specialized than the first warehouse-scale computers, and that specialization will only increase. On the frontier, we are seeing a trend toward resource disaggregation through specialized resource-specific hardware units. At the extreme, technologies like integrated silicon photonics and custom SoCs promise a fully disaggregated datacenter. System interfaces like web services and distributed operating systems were a good fit for traditional datacenters, but future physically disaggregated WSCs will require an equally disaggregated *logical* interface like serverless. In the coming chapters I move the frontier of disaggregation forward with new techniques for physical disaggregation, and new logical models to support them.

Chapter 3

Hardware/Software Co-Design Methodologies

3.1 Overview

Designing hardware is challenging. Unlike software, hardware cannot be changed after it is released (well, not much anyway). This means that designs must be extremely high performance and stable before being sent to fabrication. As a result, the hardware design process has traditionally followed a rigid waterfall development model while the tools available to designers favor low-level control over productivity. Furthermore, the high barrier to entry has led to a heavily siloed industry that rarely leverages open-source to accelerate development. This state of affairs makes research into computer architecture very challenging, particularly in an academic setting with limited resources and person-power.

Of course, hardware isn't useful without software. Architectures that don't embrace the needs of software are doomed to fail. Ideally, hardware and software teams should work together to *co-design* their systems for maximum impact. Unfortunately, hardware design has one further challenge that complicates this ideal: the need for simulation. If we are going to develop software for a new piece of hardware, we need to run it on that new hardware. The problem is that we can't just spin out a new chip over night; fabrication costs millions of dollars and takes months to complete. Instead, we rely on simulators that execute our software on a faithful model of the hardware design, a slow and resource-intensive process. This, coupled with the inherent difficulties of writing software for custom hardware, makes the co-design process difficult to achieve in practice.

Resource disaggregation presents a particularly difficult target for co-design. Disaggregation is fundamentally concerned with relatively large clusters of networked components rather than a single self-contained accelerator or system on chip (SoC). We also require long-running and complex software workloads to fully understand the behavior of any new technique. Indeed, many of my early efforts were stymied by the rigid development practices and slow evaluation methodologies that were available to me. I needed a new approach: agile hardware design. Being agile means that we can quickly modify our designs to changing requirements without sacrificing quality. In the world of software development, we can leverage standardized and open application programming interfaces (APIs), open-source software, and high-level languages to be more agile in our development. The question was: how can we get those same benefits for hardware/software co-design?

In the remainder of this section, I quickly review several ways the community at large has progressed in answering this question. Later in the chapter, I describe the tools and methodologies for agile hardware design that I developed with the Berkeley Architecture Research group to support my work on disaggregation. §3.2 goes over our SoC develop framework, called Chipyard, while §3.3 describes FireMarshal, the software workload management tool I built as part of Chipyard. Ultimately, I argue that research into new hardware designs for disaggregation will require an agile and collaborative methodology that tightly integrates hardware *and* software development processes.

3.1.1 Standard Interfaces

While designing hardware is challenging, building a functional software stack on top of a custom chip can account for over a third of total development costs [115]. Much of this effort goes toward relatively mundane tasks. Operating systems must be ported, new targets and extensions must be added to compiler toolchains, and any number of common software packages must be fixed. This is expensive and time-consuming.

The need for all this effort largely results from differences in the primary interface between hardware and software: the instruction set architecture (ISA). The solution is to develop a fully open and extensible ISA that can be used for any new project by anyone. There have been a number of attempts at open ISAs including SPARC, OpenRISC, and MIPS [287, 197, 178, 286]. While these open ISAs saw some adoption, they also carried technical and legal limitations [19]. In response, a group at UC Berkeley developed a new ISA called RISC-V that was truly open and designed to support a wide range of CPU designs though a flexible extension system [283]. RISC-V has since seen wide adoption in both academia and industry, with many high-quality open-source and proprietary implementations [116]. This widely adopted and open ISA meant that software could be ported once and re-used across a wide range of implementations.

3.1.2 Open Source Hardware

There are now a number of open-source designs for full-stack RISC-V based systems-on-chip (SoCs). These include frameworks like OpenPiton, BlackParrot, ESP, and Chipyard [27, 210, 168, 10]. These designs include complete hardware implementations of processors that support the RISC-V privileged specification [284]. They boot full operating system kernels such as Linux and support a broad range of applications. Together with additional platform-level components, these frameworks enable the design of complete SoC implementations at fabrication quality. I will refer to these concrete, synthesizable, designs as *register-transfer level descriptions (RTL)* as opposed to, e.g., analytical models of hardware.

Figure 3.1 depicts the typical components included in such a system. Open-source SoC development frameworks often provide a baseline hardware implementation and allow users to modify or add components in order to customize the SoC for a particular use-case. Evaluating such a system often requires a fully functioning software stack from firmware all the way up to user-space applications.

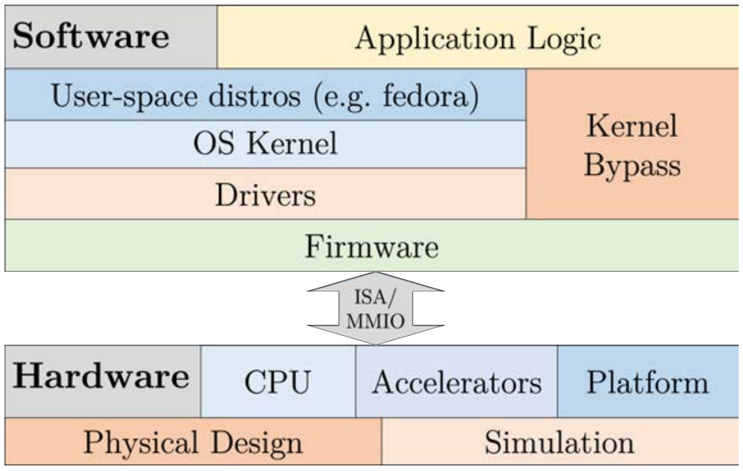


Figure 3.1: Full-stack hardware development components.

3.1.3 Software Stacks

While RISC-V ensures that software does not need to be ported to each new design, generating and maintaining a working software stack for a particular project is still challenging. The top half of Figure 3.1 shows the range of software that might be affected by custom hardware. At the lowest level, we have the firmware and boot loader, device drivers, and operating system kernel. These are usually compiled into a single boot binary that is loaded by the hardware system. Above that, we have a wide range of user-space software including utilities like networking or shells as well as user application logic. These are packed into a filesystem image that is loaded by the boot-binary. Together, these components form the *software workload* that must support the hardware platform. Projects may require multiple software workloads. For example, one workload may provide a generic interactive environment while others support automated experiments or unit tests. A change in any part of this stack may require changes to others. For example, an updated platform device may require a new software driver, or a change to the CPU boot configuration may require updates to the firmware. Likewise, updated software components may expose bugs in the hardware implementation or require new features. While a hardware project may require changes to any part of this stack, it is unlikely to require changes to *all* parts.

Developers use these software workloads to support experimentation, testing, and deployment. This *software workload lifecycle* includes specifying the workload in a shareable and repeatable fashion, building that specification into the boot binary and disk image, running it in multiple levels of simulation, and finally installing it onto the real hardware. This process needs to be repeated periodically during the development process so automated testing is also valuable.

Containers have become a standard mechanism for building and distributing workloads in the cloud and other server-side environments. Docker is a popular tool for describing and distributing these containers [175]. While there is much to learn from Docker's composable workload descriptions and highly configurable build process, it does not directly apply to hardware development. Docker only manages a Linux userspace setup and cannot modify the full software stack as needed for architectural research. It is also not designed to manage the other parts of the workload lifecycle like testing, output parsing, or simulator integration.

For lower-level control, developers can leverage Linux distributions that package most user-space components into a coherent environment. Some like Fedora or Debian target general-purpose workloads [87, 76]. For more specialized environments, there are distribution generators like Buildroot, Kickstart, or AutoYast that allow users to describe a minimal Linux environment in a configuration file [46, 161, 20]. Of particular note is Yocto which includes a flexible and composable build system called Bitbake [296].

These systems are primarily designed for system administration and deployment rather than experimentation. They do not manage user applications, experiment management, or simulator integration. For that, hardware development frameworks usually include some form of software development kit (SDK) to jump-start software development. For example, Raspberry Pi, Nvidia and Xilinx all provide SDKs for some of their products [189, 193, 305]. For RISC-V SoCs, examples include the Ariane SDK and the SiFive freedom-u-sdk [14, 94]. The SDKs integrate an embedded distribution generator, along with a default Linux kernel configuration and firmware tuned for their platforms. These SDKs are primarily targeted at producing a production software platform rather than a suite of experiments over the rapidly changing and non-standard hardware used by architecture researchers. In §3.3, I present my approach to overcoming these limitations.

3.1.4 Simulators

Open ISAs, hardware implementations, and software stacks allow us to quickly develop new systems, but we still have to evaluate them. To do that, we use a spectrum of simulators at different levels of detail and performance. On one end of the spectrum, we find functional simulators such as QEMU [216] and riscvOVPSim [226] that faithfully implement the system specification without particular concern for timing modeling. These can often be used as a reference implementation of system behavior for verification. On the other end, we have cycle-exact RTL simulators such as VCS, NCSim, ModelSim and Verilator [254, 252, 180, 272], as well as RTL hardware emulation tools such as Palladium, Zebu, and FireSim [49, 301, 134]. In between, we find functional ISA simulators such as Spike [257], as well as cycle-approximate modeling simulators such as gem5 and Sniper [263, 167].

The general trade-off is between modeling-detail and performance. While functional simulators are very fast and flexible, RTL simulation is much slower and requires complete hardware designs, but provides a higher fidelity of performance results and feature correctness. Ideally, initial software development can be done on functional simulation while slow and expensive cycle-exact simulation is only used for hardware verification and final performance evaluation. However, switching between simulators is not a trivial task, and software setup is often tightly intertwined with some simulator assumptions.

3.2 The Chipyard SoC Development Framework

There are a lot of moving parts in the hardware development stack. At the top of the stack, we have software applications, operating systems, and low-level interfaces. Below that, we need interfaces between hardware and software. These include the ISA and platform APIs. Next we need to implement those interfaces in RTL using some hardware-description language. Finally, we have to simulate those designs or convert them into wires and gates on a piece of silicon (typically referred to as VLSI). Each of these steps requires specialized tools and methodologies, but they all need to work together to create a complete chip. At UC Berkeley, we had experience with most of these steps, but their interactions were ad-hoc and driven by word of mouth. As we became more agile in our hardware development flows, this ad hoc approach became untenable. The result was a unified project, called Chipyard, that brought together all of these tools and codified the chip development process into a unified flow (see Figure 3.2). As a large collaborative project, I will leave a comprehensive description of Chipyard to our jointly published work [10]. Instead, I will use this section to quickly go over the general design process using Chipyard, with a focus on the hardware-software co-design methodology I used in my work on disaggregation.

Chipyard facilitates five key phases of the co-design process (Figure 3.3):

1. **Specification:** We begin by documenting an initial design in a semi-formal specification. We then implement the proposal in a functional model that strictly defines the expected behavior. This becomes the contract between hardware and software.

2. **Leverage Existing Designs:** Any given proposal may need to modify any part of the hardware/software stack, but it is unlikely to modify *all* parts. We therefore find an existing, known-good, implementation of a base system to serve as a starting point and reference baseline.

3. **Focused Modifications:** Once we have a base design that is functional end-to-end, we can begin implementing our proposal. Since our base design has most of what we need, our changes can be focused and will involve minimal work on unrelated components.

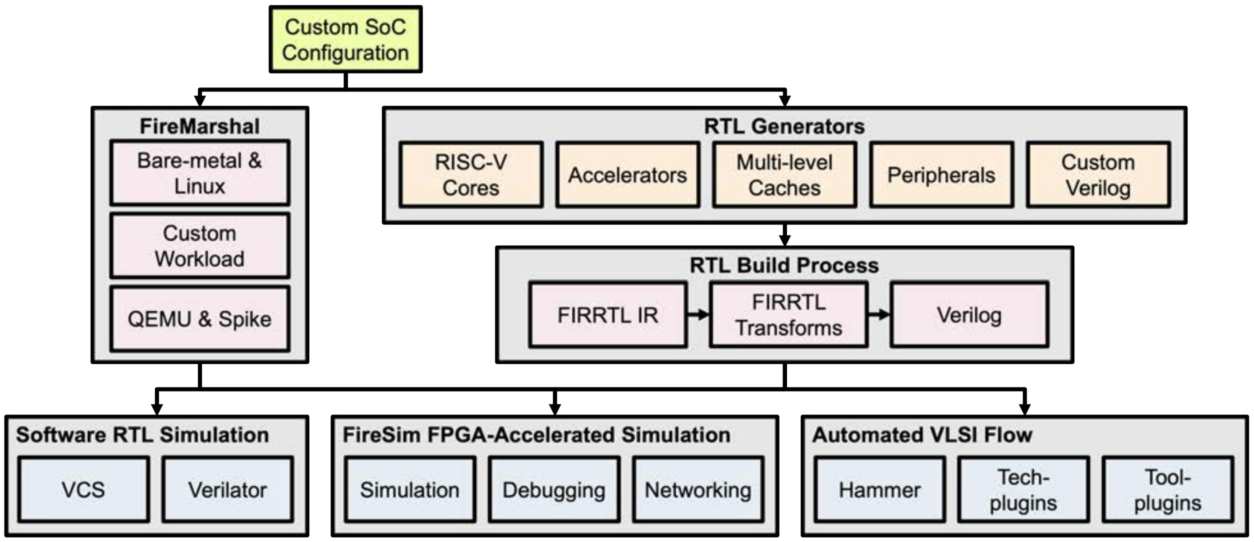


Figure 3.2: General structure of the Chipyard SoC development framework. Chipyard consists of many independent tools spanning everything from physical design and layout to software workload management.

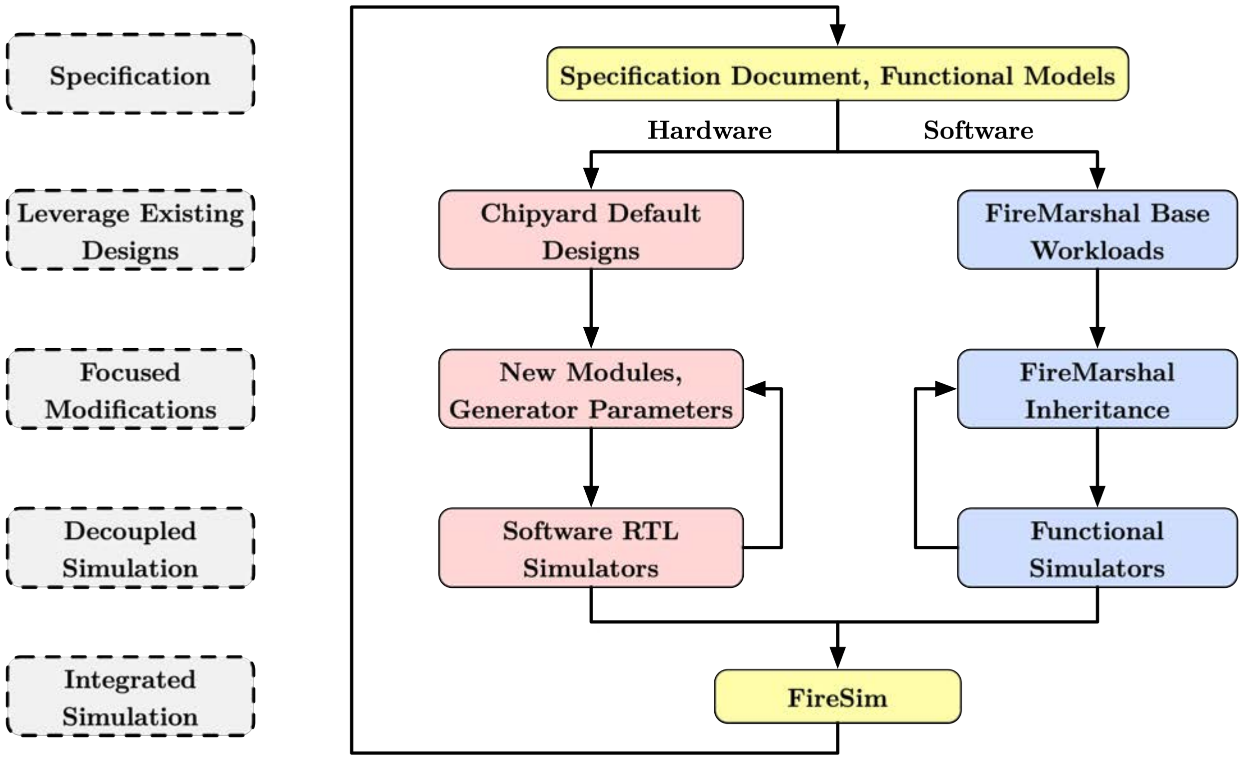


Figure 3.3: A typical agile hardware design flow using Chipyard

4. **Decoupled Simulation:** To evaluate and debug our designs, we leverage domain-specific simulators. For hardware, we are primarily interested in cycle-by-cycle behaviors of our modules or short sequences of instructions for end-to-end testing. Software-based RTL simulators give us high timing accuracy and insight for these short tests, but run too slow for end-to-end benchmarks. In software, we rely on the behavior of our hardware on complex workloads, but we are not concerned with the details of how it produces that behavior. This allows us to use functional simulators that elide timing accuracy in favor of high performance.

5. **Integrated Simulation:** Once we have confidence that both hardware and software behave correctly, we evaluate them together for correctness and performance. This step requires both high simulation speed and high fidelity. For this, we can use hardware-accelerated simulators that have higher cost and more complex deployment, but can simulate realistic software workloads in a reasonable time frame.

I now present these phases in more detail and describe how Chipyard facilitates them.

3.2.1 Problem Identification and Solution Specification

Chipyard includes a set of reasonable and high-performance designs, including end-to-end software workloads for standard benchmarks. Furthermore, projects that use Chipyard are easily shared with the open-source community. This provides a rich ecosystem of designs to evaluate when identifying new problems. A security researcher may use our out-of-order core (BOOM [58]) to find new speculative execution exploits (such as the famous Spectre exploit [142]) [101, 103, 235]. In Chapter 4, I will talk about how I used Chipyard's base designs, along with an open-source RDMA network interface to identify performance bottlenecks in disaggregated memory.

Once a problem is identified, we propose a solution. This solution might be a new hardware accelerator or microarchitectural modification. From here, we write a semi-formal specification in prose. This specification describes only the proposed changes to the base system and must include enough detail for hardware and software developers to implement the proposal. Next, this specification is codified into a functional model. In Chipyard, we typically implement this model manually using the included Spike RISC-V ISA simulator [257]. This functional model serves as the contract for both hardware and software designs. So long as both designs conform to the functional model's behaviors, we can have confidence that our designs will work together. In practice, we rarely get the functional model completely right the first time, so this process is iterative.

3.2.2 Building on Existing Designs

Chipyard includes a complete, silicon-proven, set of designs. These are tied together using a modular SoC generation framework called Rocketchip [18]. We provide a number of modules for common SoC components. This includes two CPU cores: an in-order core called Rocket, and an out-of-order core called BOOM [58]. We also have modules for network interfaces, caches, accelerators, etc. All of these components are written in the Chisel hardware-description language [22]. Chisel allows us to design our modules as *generators* rather than fixed designs. Generators allow us to parameterize our designs so that a single implementation can be instantiated with different features. For example, the BOOM core can be instantiated as 2, 4, or 8-way out-of-order. In some projects, we simply modify these parameters to explore different design points. For more complex designs, we can implement new modules in Chisel and integrate them using Rocketchip's on-chip interconnect Tilelink [251], or as an ISA extension using RoCC [21].

On the software side, we provide base workloads for general-purpose Linux distributions, as well as standard benchmarks like SPEC or Coremark [256, 96]. As with hardware, there are more software workloads available in the open-source community. These workloads are specified using the FireMarshal software-workload management tool that I developed [205]. FireMarshal allows users to inherit from these base workloads by specifying whatever application-specific changes must be made. I describe FireMarshal in detail in §3.3.

3.2.3 Correctness Testing and Evaluation

While designing new modules, we are primarily concerned with the correctness of our implementations. We require tight feedback cycles for this correctness checking since designs change rapidly and debugging often requires testing many small changes. In hardware, we use RTL-level simulators like Verilator or VCS [254, 272]. These simulators do not require any specialized resources and can run directly on our development machines. They also provide high levels of insight into our designs (e.g., individual cycle-level waveforms). Unfortunately, this convenience comes with a high performance cost. Software simulators typically run in the kilohertz and produce far too much data for evaluating long-running end-to-end benchmarks. This trade-off is acceptable in the primary implementation phase as small traces of instructions are typically sufficient for verifying compliance with the specification.

During software development, we are not concerned with the cycle-by-cycle behaviors of our hardware. Instead, we rely on *functional* simulators that faithfully reproduce the behaviors of our specification, but do not rely on any particular hardware implementation. These simulators can be very fast, often running at near-native speeds. In Chipyard, we use QEMU for simulating standard RISC-V software due to its extremely high performance [216]. For more custom designs, we use the official RISC-V ISA simulator, Spike [257]. Spike has extremely high-fidelity to the RISC-V specification and is easy to extend for new designs. FireMarshal integrates directly with these simulators to allow rapid iteration on designs and frequent regression tests.

Eventually, we need to evaluate our hardware and software designs together. This is important for correctness since our specification may be incomplete or incorrect. More importantly, we need to evaluate the performance of our design on real end-to-end benchmarks. This requires running billions of instructions on our real hardware implementation with high fidelity. Functional models are fast enough, but don't match the hardware implementation precisely. Software RTL simulators are sufficiently accurate, but simply too slow to be practical. They also lack the network models needed by research on warehouse-scale computers (WSCs) and disaggregation. Instead, Chipyard uses an FPGA-accelerated cycle-exact RTL simulator called FireSim for integrated end-to-end evaluation [134]. FireSim uses a tool called Golden Gate that takes RTL designs and converts them into a timing-decoupled model running on FPGAs [166]. It is important to note that this process does *not* produce an FPGA prototype of the design, it creates an FPGA-accelerated *simulator* of the designs. This process is able to create a simulator that runs at tens or hundreds of megahertz. FireSim instantiates these simulators on cloud FPGAs using Amazon's F1 instance type and automates much of the simulation process. FireSim was motivated, in part, by my research efforts toward a disaggregated datacenter called FireBox that required simulation of entire clusters. To do this, FireSim includes a cycle-exact network model and connects many FPGAs, each simulating one or more instances of our SoC design. We have simulated clusters of up to 1024 nodes using this approach. While FireSim is extremely high performance and accurate, it requires specialized resources (FPGAs) that may be limited and/or expensive. It also requires more effort to set up than local simulation. Fortunately, our decoupled design process allows most development to be done locally, with FireSim used only for final evaluation.

3.3 Software Workload Management with FireMarshal

As we saw earlier in this chapter, software workload management on custom hardware is a tricky problem. Chipyard and other similar frameworks have drastically improved our ability to design and simulate complex RISC-V based SoCs. These advances have greatly increased the complexity of software that can be reasonably used for evaluation; a blessing and a curse. A complete software stack needs to track the exact version of various hardware interfaces with software functionality from the firmware up to user-level applications. This increased complexity and velocity presents challenges to the management of software workloads for experimentation and research. Firstly, we must be able to rebuild and re-run our own experiments in a consistent way (repeatability). Second, we must communicate our experiments in a way that allows the community to evaluate and compare them (reproducibility). Furthermore, we would like to avoid duplication of effort within the community by reusing workloads, even as software and hardware evolve (benefaction[1]).

In this section, based on joint work with Alon Amid [205], I present FireMarshal, a software workload management system to wrangle this complexity. FireMarshal allows users to describe and share workloads in an unambiguous human and machine readable form that can be stored, version controlled, and shared. FireMarshal is included in the Chipyard

[1]The terms "repeatability" and "reproducibility" are used as defined by the ACM [16] while the term "benefaction" is derived from the work of Collberg and Proebsting [68].

framework, though it is designed to be general purpose and extensible. In §3.3.1 and §3.3.2, I describe problem of software workload management in more detail. §3.3.3 and §3.3.4 present the FireMarshal tool design and implementation. I conclude in §3.3.5 with a number of example use-cases for FireMarshal.

3.3.1 Software Workload Management Pitfalls

There are a number of common pitfalls to an ad-hoc approach to workload management. The first is *simulator compatibility*. Each simulation platform may require a slightly different configuration and care must be taken to ensure that software remains correct and faithful to the experiment when switching simulators. Another common pitfall is the generation of *magic images*: software workload artifacts that were built ad-hoc and are hard to reproduce. System configuration is challenging and error-prone, if multiple manual steps are needed there is significant room for forgotten steps or inconsistencies. Furthermore, a poorly documented build process can make experiment reproduction difficult or impossible. Finally, without additional system support, experiments may require *manual interventions*. Users need to wait for the system to boot completely before logging in and running a benchmark, and results need to be manually extracted from the serial output or disk image after a run. These interventions can introduce non-determinism in the experiment and, again, are time consuming and error prone.

3.3.2 Requirements

In contrast to the ad-hoc approach, I advocate for the use of an automated *workload management system* where the software workload life-cycle is managed automatically through standardized workload descriptions. I now identify several key requirements that a more general workload management tool should provide:

1. **Flexible Design:** Users should be able to change any part of the system, but provide only what is needed for their specific project. Reasonable and up-to-date defaults must be available for all system components.

2. **Maximal Reuse:** Workloads must be described in a way that can be shared and built upon without inside knowledge.

3. **Flexible Simulation:** It must be easy and reliable to switch between different levels of simulation while minimizing software differences.

3.3.3 The FireMarshal Tool

To address these requirements, I developed FireMarshal. FireMarshal is an open source software workload management tool for RISC-V based hardware systems development [90].

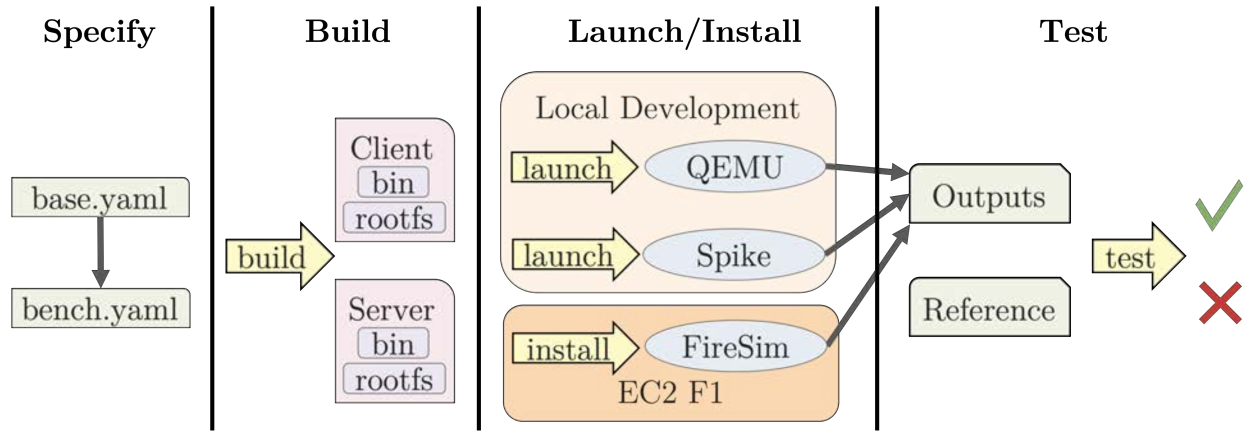


Figure 3.4: FireMarshal Workflow

FireMarshal generates workloads from machine-readable configuration files in JSON or YAML. Under FireMarshal, workloads can be tracked in a version-controlled repository and reproduced as needed. Configuration files specify a base workload to serve as a starting point, and any workload-specific changes that must be made to that base (see §3.3.4.1 for details). FireMarshal comes with several standard workloads that are configured to work on the target platform and are updated regularly to keep in sync with the evolving ecosystem. We currently supply general-purpose Fedora and Buildroot-based Linux distributions. Complex projects may create hierarchical workloads, where common options are defined once and inherited by many workloads. Most software development can occur in functional simulation on any development machine, with slow and expensive RTL simulation utilized only to drive the final performance evaluation.

FireMarshal is designed around five major phases of the workload lifecycle: **specify, build, launch, test,** and **install** (depicted in Figure 3.4). Users begin by creating a FireMarshal **specification** for their workload; they can then **build** the software artifacts (i.e., boot binary and disk image). After building the workload, users can **launch** it in fast functional simulation for **testing** and software development. Once users are satisfied with their workload, they can **install** it to a cycle-exact RTL simulator for performance evaluation. The same tests can be run on both functional and RTL simulation to ensure consistent behavior.

3.3.4 FireMarshal Design and Implementation

FireMarshal is implemented as an open source command line application along with a set of preconfigured software components. Table 3.1 summarizes the commands that FireMarshal supports. In the following sections, I describe how FireMarshal supports each phase of the software workload lifecycle.

Command	Description
`build`	Construct the filesystem image and boot-binary
`launch`	Launch this workload in functional simulation
`install`	Set up a cycle-exact RTL simulator to launch this workload.
`test`	Build and launch the workload and compare its outputs against a reference

Table 3.1: Commands supported by FireMarshal.

3.3.4.1 Specify

The workload lifecycle begins with users specifying their workload through a YAML configuration file, along with any artifacts that should be included (e.g., benchmark sources). FireMarshal provides options for workload inputs and outputs, component customization, and hooks for user scripts to run at different points in the workload lifecycle. Table 3.2 describes several common options. All options except `base` are optional.

Inheritance and Jobs A key concept in FireMarshal is *inheritance*. There are many available options for each workload, some fairly complex. To minimize repeated work, FireMarshal allows users to specify only the options that have changed relative to a `base` workload. For example, many workloads change only the `run` option to create workloads for different benchmarks while the base may include a filesystem overlay or a script for installing benchmark prerequisites.

Some simulators support multi-node simulations. In this case, several workloads are expected to run simultaneously. The `jobs` option allows users to specify multiple related workloads. Jobs are implicitly based on the top level workload description and follow all inheritance rules.

Boards and Bases FireMarshal supports multiple hardware platforms through the abstraction of a *board*. Boards encapsulate support for SoC details, peripherals, and any associated logic or quirks. Users will rarely need to define or modify a board, they should be provided by the SoC generation framework. Instead, users inherit from common *base workloads* provided by the board that abstract these details. To define these base workloads, the framework authors must provide a number of key components:

- **Linux Source:** A version of Linux known to work with the board or a link to the default version included with FireMarshal.
- **Firmware:** RISC-V systems require a supervisor binary interface (SBI) to perform low-level functions. Users may provide their own implementations of either OpenSBI [198] or the Berkeley Boot Loader (BBL) [214].

Option	Description
`base`	Start from a pre-existing workload
`overlay/files`	Files to include in the image
`host-init`	Script to run before building (e.g. cross-compile)
`guest-init`	Script to run once on the guest (e.g. install packages)
`run/command`	Script to run every time the image boots (e.g. default experiment)
`outputs`	Files to copy out of the image after an experiment
`post-run-hook`	Script to run on the output of the experiment (e.g. parse or format results)
`linux`	Linux customization options including Linux source directory, kernel configuration options to modify, as well as any needed kernel module sources
`firmware`	Firmware-related options including choice of firmware and build options.
`spike`	Custom Spike binary to use
`spike/qemu-args`	Additional arguments to pass to functional simulators
`jobs`	Additional, related images to build (each node of a networked workload)

Table 3.2: Common FireMarshal configuration options.

- **Drivers:** If the board includes any additional devices such as a network or disk interface, the user must include the needed Linux drivers. Drivers will automatically be built and loaded by FireMarshal.
- **Base Workloads:** A board must include base workloads for supported distributions.

3.3.4.2 Build

The next step in the workload lifecycle is to build the workload. A FireMarshal build produces a bootable binary and a filesystem image (Figure 3.5). The boot binary includes the firmware, Linux kernel, and an embedded filesystem (*initramfs*) containing platform drivers and other early-boot code. In some cases, users may wish to produce a workload that does not involve a disk device. In this case, they specify the `--no-disk` command line option, which causes the disk image to be embedded in the initramfs. This process happens transparently and does not require further user intervention.

Build Phases FireMarshal goes through a number of steps during a build, although not every step is required for every workload:

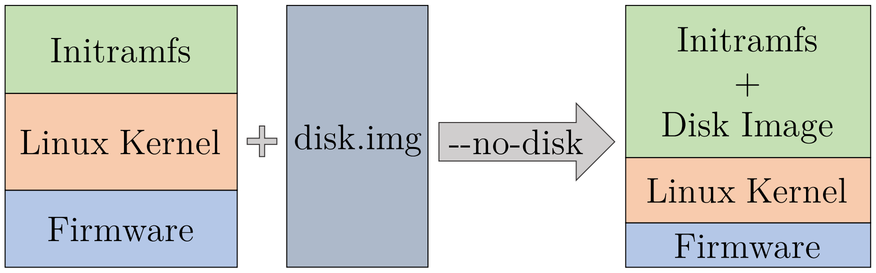


Figure 3.5: **Outputs of the build command** By default, a complete bootable binary and a disk image are produced. For diskless builds, users provide the `--no-disk` option, in which case the disk image is embedded in the Linux embedded ramdisk (initramfs).

1. **Configuration:** The first step is to read the workload configuration file and any potentially related configurations. FireMarshal employs a search order similar to the `$PATH` variable in a Unix shell to locate workloads. Parent workloads are parsed recursively, with children inheriting options from their parents (and overriding as needed).

2. **Build Parents:** The build process from this step forward is performed recursively to produce filesystem images for all parents.

3. **host-init:** If the workload includes a host-init script, this is run before proceeding to ensure that any generated artifacts are available in future steps.

4. **Boot Binary:** If the user has hard-coded a boot binary, the following steps are skipped. If the child workload would not generate a different binary than its parent, FireMarshal simply makes a copy of the parent's binary and skips this step.

   a) **Final Linux Configuration:** To form the final Linux configuration, FireMarshal begins with the RISC-V default configuration. If needed, users can provide Linux kernel configuration *fragments* that contain a list of options to change in the default configuration. The use of configuration fragments makes workloads more portable between kernel versions.

   b) **Kernel Module Generation:** With a valid kernel configuration, any needed kernel modules defined in the workload can now be built. This includes system-provided device drivers, as well as user-provided kernel modules.

   c) **Generate Initramfs:** To load drivers as early as possible, and to provide a mostly workload-independent boot phase, FireMarshal generates an initramfs as the first-stage init. This initramfs loads both system and user-provided kernel modules.

   d) **Linux Compilation:** The full Linux kernel can now be compiled with a reference to the initramfs to embed.

   e) **Firmware:** The desired firmware is compiled and linked with the Linux binary. At this stage, the boot binary is complete.

5. **Disk Image:** As with the boot binary, users may provide a hard-coded disk image in which case the following steps are skipped.

   a) **Copy Parent Image and Add Files:** FireMarshal makes a copy of the parent's disk image and then copies over any files from the `file` or `overlay` options.

   b) **guest-init:** At this stage, we have a bootable (albeit incomplete) workload. FireMarshal now configures the workload to run the guest-init script and boots it in QEMU. This script is run exactly once.

   c) **Boot Command:** The final step in filesystem generation is to configure the workload to run user-provided code from the `command` or `run` option on every startup. This is done by inserting a new step in the Linux distribution's init system.

6. **Initramfs-Embedded FileSystem:** As shown in Figure 3.5, users may provide the `--no-disk` option to FireMarshal to eliminate the need for a disk device. To do this, FireMarshal runs the build process as described above, but recompiles the kernel with the generated disk image as its initramfs payload.

As this process can be quite time consuming, especially for workloads with deep inheritance hierarchies, FireMarshal uses a dependency tracking system to avoid unnecessary rebuilding [237].

3.3.4.3 Launch

After building, the `launch` command runs the workload in functional simulation. Serial inputs and outputs are presented to the user interactively and logged to a file for later analysis. For workloads with a `command` or `run` option, the user does not need to interact with the simulation. When the simulation completes, FireMarshal copies any output files and the serial port log to an output directory. The `post-run-hook` script is run against this output to produce final results.

3.3.4.4 Test

While the `launch` command is primarily used for interactive debugging and development, FireMarshal supports hands-off testing with the `test` command. This is useful for running suites of automated tests like those seen in continuous-integration (CI) workflows. The test command builds and launches the workload, and then compares the outputs against any provided reference outputs. A complete comparison of outputs is not typically appropriate as there may be irrelevant or non-deterministic output (e.g. time stamps). Instead, FireMarshal is able to clean outputs and allows the reference to contain only a subset of the expected output. A test that produces that subset somewhere in its output is considered a success. Workloads with more complex success criteria can use the `post-run-hook` option to perform custom analysis of outputs.

3.3.4.5 Install

Once a workload passes functional simulation, users may wish to run it against a cycle-exact RTL-level simulator. Unlike functional simulation, RTL-level simulators require hardware-specific configuration and build processes that are out of scope for a workload management tool like FireMarshal. Instead, FireMarshal provides the `install` command to convert the workload specification into a valid configuration for the RTL-level simulator. From there, users interact with the simulator normally to launch the workload. After a simulation, users can verify the outputs using the `test` command with the `--manual` option to compare outputs as if FireMarshal had run the workload. It is important to note that the workload outputs are not modified in any way between the `launch` and `install` commands; the exact same artifacts are run on both simulators.

3.3.5 Case Studies

I now review a few examples of how I have used FireMarshal in my own research as well as some community use cases.

3.3.5.1 The Page Fault Accelerator

For this case study, I show how I used FireMarshal to support research into using OS paging to implement physical memory disaggregation. I will cover this project in detail in Chapter 4, but I give an overview of it here. Figure 3.6 gives a high level view of this *page fault accelerator (PFA)*. The PFA was designed to improve the performance of systems that use remote memory as a swap device (e.g., Infiniswap [104]) by handling the basic remote memory lookup and fetch in a new hardware module embedded in the MMU. The complex paging logic in the OS could then be deferred to an asynchronous background thread. The OS interacted with the PFA through several memory-mapped queues and special page table entry values. Similar to regular RDMA, local memory regions were registered with the PFA for fetched pages. The PFA directly interacted with the network interface through its exposed queues, much the same way an OS driver would.

My collaborators implemented the accelerator itself using Chipyard in a few hundred lines of Chisel while I focused on the software components. Since Chipyard provided a complete base system, including an RDMA-capable network interface, the hardware components required relatively little engineering effort. However, the kernel modifications to support the accelerator were extensive and complex. Beyond the kernel, user level services like systemd and cgroups required careful configuration and integration with experimental procedures. This project required a number of software tasks:

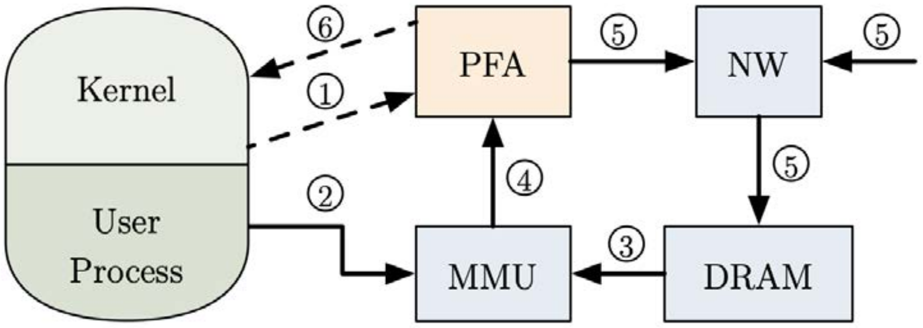


Figure 3.6: **Block diagram of the PFA** The kernel asynchronously provides free physical pages to the PFA ①. On a page fault ②, the MMU consults the page table ③ and requests any remote pages from the PFA ④. The PFA initiates an RDMA operation from the network adapter ⑤. The kernel can now asynchronously request a list of fetched pages for bookeeping purposes ⑥. The critical path for a remote page fault (steps ②-⑤) is handled synchronously in hardware while slow kernel interactions (steps ① and ⑥) are moved off the critical path.

Bare Metal Unit Tests To verify software drivers and the hardware implementation, I implemented a golden model of the PFA in the Spike functional simulator. The golden model exposed all software-visible interfaces and emulated remote memory. Low-level tests were implemented either completely bare metal or in the RISC-V proxy kernel [214]. These tests were critical for debugging hardware implementation issues and served as a reference for the specification. In my FireMarshal workload configuration, I included a reference to the modified simulator (using the `spike` option) and a script to cross-compile the benchmarks (using the `host-init` option). I ran this using the `launch` command and debugged interactively until I was satisfied that it worked correctly. The serial port output was saved as a reference output (using the `testing/refDir` option) and used for regular automated tests (with the `test` command). This same workload could then be run on FireSim to verify the hardware implementation using the `install` command. This test was revisited periodically as the hardware specification evolved, or as new corner-cases were identified that required additional unit-tests.

Linux Unit Tests The most significant engineering challenge in the PFA project was to modify the Linux kernel to asynchronously process page faults. I also modified the default Linux kernel build configuration to enable certain swapping-related features. For basic unit testing, I used the Buildroot base workload. Buildroot could boot quickly and minimized the amount of extra code running in the system. I began with `pfa-base`, a base workload that would handle the common setup tasks (Listing 3.1). Individual tests and benchmarks inher-

```
{
  "name" : "pfa-base"
  "base" : "buildroot",
  "host-init" : "cross-compile.sh",
  "linux" : {
    "source" : "pfa-linux/",
    "config" : "pfa-linux.kfrag",
  },
  "overlay" : "pfa-test-root/",
  "spike" : "pfa-spike"
}
```

Listing 3.1: Base workload for PFA Linux unit tests. This workload was written once and re-used by all Linux-based tests and experiments.

```
{
  "name" : "latency-microbenchmark",
  "base" : "pfa-base",
  "post-run-hook" : "extract_csv.py",
  "jobs" : [
    {"name" : "client",
     "linux" : {"config" : "pfa.kfrag"}
     "command" : "latencyTest.sh"
    },
    {"name" : "server",
     "base" : "bare-metal",
     "bin" : "serve"
    }
  ]
}
```

Listing 3.2: An example microbenchmark workload. The microbenchmark has one Linux-based job for the client benchmark, and a bare-metal job to serve remote memory. The jobs will be instantiated as network nodes in FireSim simulation. There were many workloads similar to this for each unit test or experiment.

ited from `pfa-base`, typically adding only a Linux configuration fragment and a `command` option to run a particular benchmark. Listing 3.2 shows an example for one particular benchmark, a microbenchmark that measured the latency of each step in a remote page fault. This workload also included a `post-run-hook` option that automatically extracted and formatted the experiment results into a CSV format. I could then inspect those results manually or produce figures for publication. I used this process to generate the results in §4.2. Note that there were many workloads similar to `latency-microbenchmark`, but only one `pfa-base`.

The first step in developing the kernel modifications was to create a non-accelerated baseline by emulating the PFA's behavior in the regular page fault handler. These modifications were non-trivial and introduced complex, non-deterministic, bugs. QEMU allowed me to run long-running tests in a reasonable time frame, as well as providing an integrated GDB server for interactive debugging. Once I was satisfied with the emulated behavior, I introduced the real hardware driver and ran the tests against my Spike golden model. The only change required in the workload was a one-line Linux configuration fragment to enable the PFA driver. This meant that the experimental setup and test parameters were identical between the two simulators, giving me confidence that any errors were due only to the driver change. When everything worked in both Spike and QEMU, I could run the unmodified workload on FireSim for final verification. Since the software had been verified against the golden model, and the golden model had been verified with the bare-metal unit tests, I could narrow down any errors quickly. Since the process of targeting different simulators was automated, there was minimal room for human error.

End-To-End Benchmarks Once I had confidence that the system operated correctly, I was able to evaluate the PFA against end-to-end macro-benchmarks. Some of these real-world applications had many dependencies that would be difficult to fulfill manually as required by Buildroot. Instead, I leveraged the package management system of Fedora to install dependencies at build time using a `guest-init` script. While more full-featured, Fedora took significantly longer to boot and introduced hard-to-debug features like asynchronous systemd services.

The workload description process was similar to that of the Buildroot unit tests, but I additionally included a `post-run-hook` option to automatically process experimental results from the serial output into CSV files for analysis. This ensured that experiments could be re-run by myself or external users and processed in a consistent way. The FireMarshal workload served as unambiguous documentation of my experimental procedure for reproducibility and comparison.

Repeatability, Reproducibility, and Benefaction While developing these workloads, I manually inspected results and interactively debugged any issues. Once they were stable, I used the `test` command to re-run the test workloads after any major changes. The end-to-end benchmarks similarly supported an automated workflow with the `launch` and `install` commands. These served as the core artifacts used for evaluation by myself and others in the community that wanted to reproduce my results.

3.3.5.2 Benchmarking: SPEC2017

Not every research project requires custom software for evaluation. Changes to a branch predictor or cache design are best evaluated using standard benchmarks. In this section, I describe how I used FireMarshal to provide one common benchmark used in the architecture community: SPEC2017 [256]. SPEC provides a number of scripts for interacting with the benchmark, while tools like Speckle [59] simplify the process of cross compiling for new architectures. However, having the binaries alone is not sufficient for a benchmark. Users still need to invoke and measure the benchmarks in a consistent way as well as compile and format results. Listing 3.3 shows one example workload for the intspeed benchmark suite.

In this section, I describe an experiment to compare two different branch predictors on the Berkeley Out-Of-Order Machine (BOOM [58]) using the intspeed benchmark suite from SPEC2017. In one case, I use an older branch predictor from BOOM v2 based on Gshare [172], in the other I use the more recent TAGE-based predictor [304, 243]. Switching between these configurations was a simple matter of modifying the BOOM generator's parameters in Chipyard.

In the general case, SPEC does not require changes to system software and simply inherits from the default Buildroot environment. This means that the SPEC workload will transparently receive any updates to the built-in workloads and will be portable across many boards and versions. Users are free to copy this workload description and change the base if their particular example requires additional configuration. No changes were needed for this branch-predictor experiment. Cross-compilation of the benchmark is provided by Speckle (in the `host-init` option), while the FireMarshal workload marks the Speckle outputs as an `overlay`. For each benchmark, the run script will place results in `/output`, so FireMarshal is instructed to retrieve these after the workload finishes running (using the `outputs` option).

Each benchmark in the suite is independent and can run in parallel. I exploit this in the workload by specifying 10 jobs, one for each benchmark. Each job differs only in the `command` option, specifying which benchmark to run. When installed to FireSim, each job is instantiated as a node in the simulated cluster and run in parallel. This optimization reduced the runtime for the experiment from about two weeks to roughly two days.

Once the workload has finished, the workload passes the results through a `post-run-hook` script that combines all results into a CSV file (Listing 3.4), as well as plotting a simple diagram for quick reference (omitted for brevity). Figure 3.7 shows the combined output of the two experiments from the result CSVs.

I developed this workload entirely on a cheap local machine using QEMU and without regard for the eventual branch-prediction experiment. I ran it for the first (and only) time on cycle-exact simulation to gather the final performance numbers on real RTL. Since FireMarshal ensures that identical inputs are run on both functional and cycle-exact simulations, I had confidence that the workload would run correctly the first time.

```
{
  "name" : "intspeed",
  "base" : "buildroot",
  "host-init" : "speckle-build.sh intspeed ref",
  "overlay" : "overlay/intspeed/ref",
  "rootfs-size" : "3GiB",
  "outputs" : ["/output"],
  "post-run-hook" : "handle-results.py",
  "jobs" : [
    {
      "name": "600.perlbench_s",
      "command": "./intspeed.sh 600.perlbench_s --threads 1"
    },
    ...
    {
      "name": "657.xz_s",
      "command": "./intspeed.sh 657.xz_s --threads 1"
    }
}
```

Listing 3.3: Workload for the intspeed benchmark suite from SPEC2017. In total, there are 10 jobs, one for each benchmark in the suite. Jobs are able to run in parallel in FireSim.

```
name,RealTime,UserTime,KernelTime,score
600.perlbench_s,1428.54,1428.0,0.43,1.24
...
657.xz_s,3034.63,2999.81,34.63,2.04
```

Listing 3.4: Example CSV output of the spec2017_intspeed workload for the TAGE configuration.

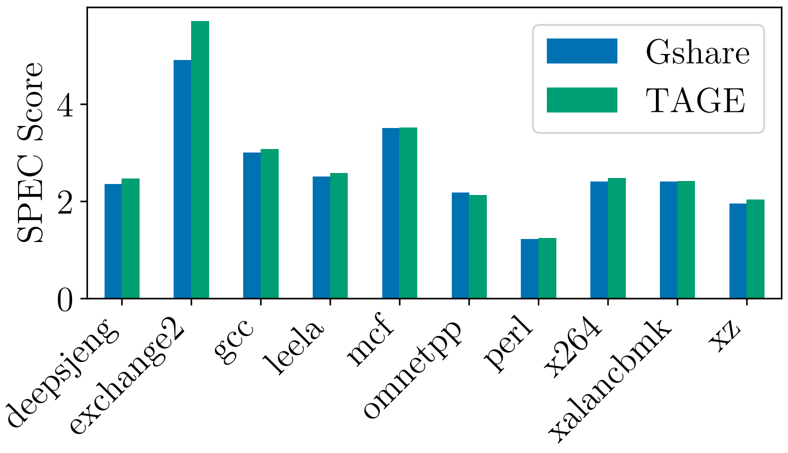


Figure 3.7: Combined graph output of the spec2017_intspeed workload. A similar graph is generated automatically for each experiment while the combined graph can be generated manually using an included script.

User Experience Most users do not need to look at the workload description. It was written once and can be reused by anyone without modification. A typical user would run the SPEC workload with the following steps:

1. **Install SPEC:** Since SPEC is closed-source software, I am unable to automate installation. Users must first acquire and install the SPEC benchmark suite sources and a license to use them.

2. **Download the FireMarshal Workload:** The FireMarshal workload can be cloned from a public git repository [255].

3. **Build the Workload:** Once SPEC is installed, FireMarshal can build the entire workload suite with one command: `marshal build intspeed.yaml`.

4. **Install the Workload:** Once built, the workload could be run in functional simulation with the `launch` command, but this is not typically needed since users do not need to do any software development. Instead, users will typically have FireMarshal create RTL simulator-compatible configuration files using `marshal install intspeed.yaml`.

5. **Run the Simulation:** Users now interact with their RTL simulator as usual, providing their hardware configuration and any other simulation parameters they wish. When the simulation completes, the simulator will provide an output directory containing the benchmark results as generated by the post-run-hook script (see Figure 3.7 and Listing 3.4).

Other than acquiring the licensed SPEC suite itself, I did not need to interact with any target software in order to run the branch-prediction experiment. All that was required was to generate my desired hardware configurations (the feature I actually cared about); the software "just worked". Furthermore, results were recorded in a standard and reproducible way. If I were to add a new branch predictor in the future, I could have confidence in my experimental setup to compare against previous results without needing to re-run them. Most importantly, now that the workload has been implemented, it is freely available for anyone to use or improve without repeated effort.

While I describe SPEC here, there are other similar benchmark workloads already available including CoreMark [96] and the ONNX-runtime deep learning framework [99, 195]. As new benchmarks are ported or developed, they too can be shared with the community in a similar fashion.

3.3.5.3 Education: Computer Architecture and Engineering

Educational settings are notoriously sensitive to consistency and reproducability of results. As computer science classes scale to a large numbers of students, mass assignments and automated grading are becoming necessities in many university courses. However, reproducability is often extremely sensitive to software versioning and simulator compatibility. A simple change in the Linux kernel version can dramatically change performance characterization results, which would be reflected in various student assignment submissions.

Furthermore, we would like students to invest their time in the educational objectives of characterization and measurement rather than spending the majority of their time on setting up environments and boiler-plate setup procedures.

While system environment platforms such as Docker or Vagrant provide a solid platform for systems-oriented classes, they are insufficient for hardware-simulation classes that require support for a broader set of configurations and cross-compilation. In 2020, FireMarshal was used in an advanced graduate and undergraduate class at UC Berkeley on the subject of hardware for machine learning [11]. As part of this class, students had to optimize tiled convolution and matrix multiplication implementations for an RTL implementation of a machine learning accelerator integrated into a RISC-V SoC. The optimizated software implementations were to be used as a library within deep neural-network (DNN) inference applications using the ResNet-50 and MobileNet DNN models. Figure 3.8 depicts the student workflow for this assignment.

To enable students to integrate their optimized libraries with the DNN inference applications, the course instructors used a FireMarshal workload definition. This enabled students to focus their time on the development of their library implementations rather than spending it on setting up their testing environment on various iterations and platforms.

As part of their development process, students initially developed their implementations using the Spike functional simulator, and then performed measurements using FireSim. By using the same FireMarshal workload definition, students were able to take advantage of the portability of FireMarshal workloads across different RISC-V simulation platforms.

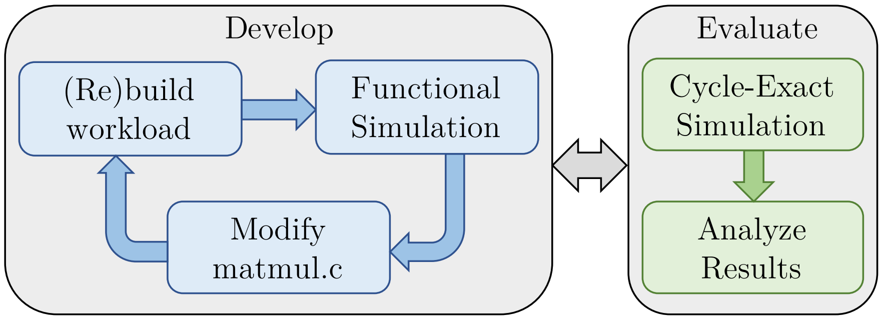


Figure 3.8: **Student Workflow** Students were asked to tune a matrix multiplication routine for a particular deep learning accelerator. The course staff provided a FireMarshal workload as a starting point. Students used fast and inexpensive functional simulation when developing their code, while slow and expensive cycle-exact simulation was only used to evaluate performance.

Thanks to the determinism of FireSim simulations, and the reproducability of FireMarshal workloads, students were able to obtain repeatable results down to an exact cycle-count of each executing application and course staff could reproduce these results for grading purposes.

3.3.5.4 Other Use Cases

I now briefly summarize additional real-world use cases:

Centrifuge Centrifuge is a tool for design-space exploration of accelerators using high level synthesis [117]. Centrifuge automatically generates several candidate accelerator designs and their associated unaccelerated baselines. I designed a common base FireMarshal workload that provides kernel and user-space modifications to support accelerator interfaces. I also provided scripts to generate per-benchmark base workloads from the accelerator descriptions. Users could use these base workloads to develop any final benchmark workloads they required. FireMarshal's inheritance mechanism and clear YAML interface made this workflow simple to implement.

Keystone Keystone is a secure enclave for RISC-V based systems [150]. Unlike other hardware enclaves, most of Keystone is implemented in the firmware and operating system. Keystone provides a FireMarshal workload that includes these changes. Enabling the enclave is as simple as switching the `base` option in a workload from the board default to `keystone-base.yaml`.

Post-Silicon Bringup FireMarshal is also being used to support post-silicon bringup of experimental SoCs. FireMarshal's configuration system and *board* primitives allow users to minimize image sizes to fit in limited on-board memories and to automate host-device interface configuration. Researchers can re-use the diverse range of existing FireMarshal-based benchmarks. The ability to run software binaries set up in an identical manner on both functional simulation and the test board is a valuable debugging capability when dealing with potentially faulty hardware.

3.4 Conclusions

This chapter has focused on tools and standards rather than grand insights into computer architecture. While these may seem like simple "nice to have" features, those grand insights come slowly, or not at all, when the community fails to be productive. More than just productivity, the computer science community, like other scientific domains, is facing a reproducibility crisis [68, 121, 144, 274, 232]. To quote Krishnamurthi and Vitek: "Science advances faster when we can build on existing results, and when new ideas can easily be measured against the state of the art" [144]. Tools that build on common designs, share their outputs openly, and automate the process of experimentation can move our community forward at an unprecedented rate. Indeed, the agile co-design methodology I described in this chapter enabled much of the work I describe in upcoming chapters.

Chapter 4

Physical Disaggregation: Memory

In Chapters 1 and 2, I introduced the concept of physical disaggregation. In this chapter, I explore this topic in more detail, with a particular focus on disaggregated memory. Memory is a crucial resource in distributed computing and it has an out-sized impact on how we structure our applications. Memory capacity limitations are central to algorithms for distributed databases, linear algebra, simulators, and almost any other data-intensive distributed application. The problem with memory is that it is difficult to share or reallocate. While CPUs can be time sliced and networks can interleave many flows, memory must be allocated exclusively for the duration of the job. Not only is it hard to structure applications around this limitation, it's hard to implement them correctly, even within that structure. Exceeding a CPU allocation simply results in the application being throttled, a performance concern. In contrast, exceeding available memory by even one byte results in a crash. To avoid this, ysers typically request more resources than they actually need [149, 224]. These properties make disaggregating memory particularly appealing as we would no longer be limited by discrete resource pools and can make far more flexible placement decisions.

We have already seen the benefits that disaggregation can bring to data management. Storage systems have been disaggregated for many years in the form of file servers and block storage appliances [188, 77, 25]. Databases and object stores also allow applications to use remote resources. While this past work has proven the potential of disaggregation, translating it to memory is not straightforward. Disks have access latencies in the milliseconds and tend to be accessed only intermittently. Memory is accessed in hundreds of nanoseconds and experiences a wide range of access patterns. Where network latencies are trivial compared to disk accesses, they are orders of magnitude slower than memory. Disaggregating memory therefore presents a true test of physical disaggregation.

I begin the chapter with an overview of the design space of memory disaggregation techniques (§4.1). I then describe one particular approach I've taken that uses demand paging to transparently present remote memory to applications (§4.2). Finally, I present a process checkpoint/restart system I developed that exploits remote memory to provide fault tolerance (§4.3).

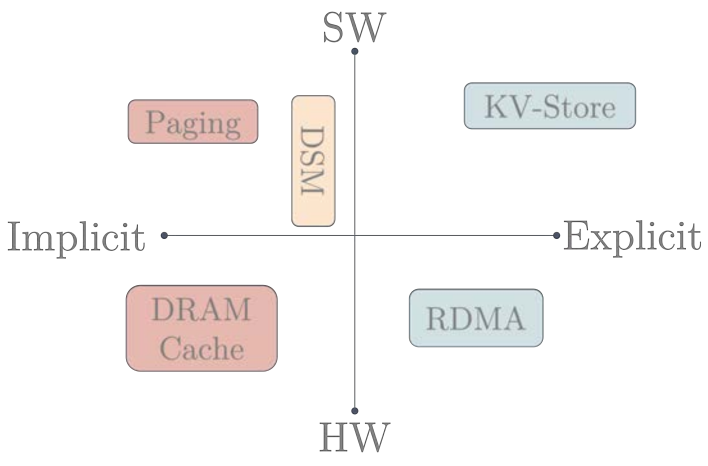


Figure 4.1: Two major dimensions of memory interface design. I place a number of common techniques in this design space.

4.1 The Memory Disaggregation Design Space

The core concept of disaggregated memory is straight-forward: allow any process to access any memory in our warehouse-scale computer (WSC). If we try to get more specific than that, definitions become more challenging. Different people have widely varying conceptions of what it means to access remote memory and how systems should be implemented to provide this access. In this section, I bring structure to that question by presenting several dimensions of disaggregated memory interface design. I describe these dimensions and list a number of additional desiderata in §4.1.1. Next, I map several existing and proposed interfaces into this taxonomy in §4.1.2. Finally, in §4.1.3, I explore the implications of this structure on future research into disaggregated memory.

4.1.1 Dimensions of Memory Disaggregation

I begin by exploring two key dimensions that allow us to compare a wide array of techniques. Figure 4.1 places some common techniques in this space.

4.1.1.1 Explicit $\leftrightarrow$ Implicit

The `Explicit/Implicit` dimension describes the level at which users must reason about local and remote memory resources. A fully `implicit` interface would require no application changes relative to a single-node aggregated implementation to operate *correctly*, though performant code may still require modification. In contrast, fully `explicit` interfaces may require significant re-writing of applications to achieve both correctness and performance.

Implicit interfaces typically resemble caches. The system is automatically moving data from remote to local memory on-demand, and evicting old data to remote memory. We can borrow terms from the computer architecture community to further refine our understanding of a particular system. For example, caches can be write-back or write-through, inclusive or exclusive, coherent or incoherent. This approach also carries the well-studied limitations of caches such as requirements on temporal or spatial locality, maximum working set size, and metadata size vs block size. One effect of this similarity is that high-performance applications are already written to take advantage of caches, and many of the same optimizations will translate directly to the disaggregated memory setting. In general, users should find the `implicit` interface to be familiar.

Explicit interfaces more closely resemble scratchpads. These systems require explicit movement of data from remote to local memory and vice versa. One implication of this strategy is that fully explicit systems require users to track their memory usage to avoid overflowing their local allocation; size miscalculations are a *correctness* issue. Scratchpads allow maximum flexibility for applications, allowing complex application-specific prefetching patterns and eviction policies. These interfaces can also be more efficient than `implicit` approaches by avoiding excessive cache metadata or stranded resources. Distributed systems and data-intensive applications are likely to support `explicit` interfaces well as these systems already must account for limited local memory. Despite this, management of local memory remains a significant challenge.

It is important to note that this is not a strict dichotomy, but rather a spectrum of designs. Traditional hardware caches are fully `implicit`, but OS-managed paging allows some input from applications (e.g., mmap, pinning, etc.). One could also imagine a system that is `explicit` by default, but degrades gracefully to `implicit` as memory pressure increases.

4.1.1.2 Hardware Assisted ↔ Software Managed

Orthogonal to the `Explicit/Implicit` dimension is the level of hardware assistance provided by the remote memory interfaces. A fully `hardware assisted` system would perform all performance critical operations in hardware. A fully `software managed` system would use only the barest possible hardware interfaces to manage memory.

Hardware Assisted systems can provide orders of magnitude performance improvements over software alone, particularly for latency-oriented tasks. However, they imply significant adoption barriers and may become outdated as systems evolve. A more subtle concern with hardware support comes from Myer and Sutherland's concept of the "wheel of reincarnation" [185]. In short, as we add sophistication to a hardware accelerator, we may end up effectively re-inventing the CPU. Hardware sophistication does not come for free. For example, smart network interface cards (NICs) often contain CPUs and non-trivial memory with their associated power and area costs. This can complicate analyses of total cost of ownership.

Software Managed interfaces benefit from flexibility and ease of deployment, but can suffer from performance limitations. While careful systems engineering can overcome some of these limitations without introducing new hardware, these solutions may require fragile tuning and complex control flow [154, 129, 107].

Again, there are appealing points along this axis that compromise between the two extremes. Hardware has the potential to introduce a much simpler interface by offloading latency critical tasks to dedicated compute elements while maintaining a rich software interface for less critical tasks. Amazon's Nitro system on chip (SoC), for example, accelerates complex network virtualization tasks but interfaces with guests running traditional operating systems [156].

4.1.1.3 Other Dimensions

In addition to the two main design dimensions, there are several other critical properties that any disaggregated memory interface must specify.

Durable $\leftrightarrow$ Ephemeral Because remote memory is in a separate failure domain from node-local memory, its lifetime can be managed independently from any particular process. Varying levels of durability can be provided, from fully ephemeral systems like memcached [174], to atomically durable like ramcloud [234]. Intermediate points, such as Pocket, are also available [141].

Isolated $\leftrightarrow$ Fate Shared Related to durability, independent failure domains allow a system designer to choose the extent to which a failure will propagate. A totally `isolated` memory system will remain available regardless of failures in any compute node. This behavior may be desirable for checkpoint/restart or systems with transactional semantics. However, we may choose to crash dependent compute nodes when their memory fails and vice versa (i.e., *fate share*). There has been work showing how to track dependencies dynamically and execute flexible fate sharing policies [53].

Private $\leftrightarrow$ Shared Finally, disaggregated memory can be connected to multiple compute nodes simultaneously. A fully `shared` system would allow fine-grained access to any memory location, by any compute node, at any time. In contrast, a fully `private` interface would allow no sharing at all, typically implying some level of ephemerality. In addition to durability and fate-sharing, shared memory must specify the access granularity in both time and space. For example, does the sharing occur at the level of bytes, pages, or some higher-level object? Must sharing occur in phases or can it occur simultaneously? The choice of consistency model can have significant implications on achievable scale and performance.

4.1.1.4 Other Requirements

In addition to the preceding dimensions, any new disaggregated memory will need to address at least the following questions:

1. **Naming:** How do tasks address remote memory? In the case of (`ephemeral`, `private`) data, this may be simple. Systems that support `sharing` and/or `durability` may impose non-trivial requirements on naming.

2. **Coherency and Consistency:** If `sharing` is supported, data may exist in multiple locations simultaneously and require special handling.

3. **Security/Fairness:** Disaggregation increases the degree of sharing in a system. A new design must include mechanisms to protect applications from both malicious and accidental interference.

4.1.2 Common Approaches to Disaggregated Memory

I now survey this design space by describing a number of established and recently proposed memory interfaces.

4.1.2.1 NUMA (`Implicit`, `HW Assisted`)

Non-uniform memory access (NUMA) architectures partition memory resources across several compute nodes such that all memory can be accessed directly with loads and stores, but access latencies are not uniform. Some NUMA systems include hardware services to aid in page migration to mitigate this effect [146]. Recent projects have extended NUMA to include disaggregated memory nodes that do not include CPUs [153] while others leverage cache coherence hardware to make memory movement decisions [51].

The `implicit` interface presented by NUMA systems is appealing because they appear to software as a single, large memory. Because they are `hardware assisted`, they can also offer memory access latencies on the order of hundreds of nanoseconds. However, this performance and tight coupling limits scalability. The largest NUMA systems can scale to hundreds of nodes and tens of terabytes of memory [244, 179, 3], but typical systems support only a few terabytes and less than ten nodes due to poor scaling in cost and power. These scalability and flexibility limitations are common in `(implicit, HW)` systems.

4.1.2.2 RDMA (`Explicit`, `HW Assisted`)

Remote direct memory access (RDMA) systems are similar to NUMA in that memory resources are partitioned among several compute nodes so that memory is always local to someone [228, 222]. The difference is that while NUMA systems typically expose an `implicit`, cache-coherent load-store interface to both local and remote memory resources, RDMA uses an `explicit` put/get interface to access remote memory. Typically, this service is provided through the network interface (`hardware assisted`). The `explicit` interface allows RDMA systems to scale beyond NUMA to thousands of nodes and petabytes of memory, but this comes at the cost of slower remote memory access performance and a more complex interface to applications [79]. RDMA can also be enhanced with location transparency, consistency and coherency models, or persistence [50, 280, 7, 163]. StRoM allows for more complex access patterns with features like indirect memory lookup or data shuffling [250].

Typical of `hardware assisted` systems, RDMA devices are costly and primarily deployed in supercomputing environments, though recent Ethernet-based implementations have made them increasingly accessible [228].

4.1.2.3 Memory Semantic Fabrics ($*$, `HW Assisted`)

A new class of interface has recently been introduced; the memory semantic fabric. A memory semantic fabric abstracts memory into a simple load-store interface rather than technology-specific protocols. These `hardware assisted` interfaces are tightly coupled with the CPU, often loading memory directly into local caches. Unlike traditional NUMA, this abstraction enables heterogeneous memory technologies in flexible topologies. From the OS perspective, memory thus becomes an `explicit` first-class citizen on a memory-optimized interconnect and can be exposed to users as such. However, the OS may choose to make the interface `implicit`, typically through virtual memory techniques. Such interfaces promise to allow for greater scalability and flexibility than NUMA, while providing a less complex interface than RDMA. There are several commercial and academic projects developing cache-coherent interconnects for integrating accelerators and memories within a rack [57, 290, 190, 157, 98, 248].

4.1.2.4 Paging and Page Migration (`Implicit`, `SW Managed`)

While memory semantic fabrics and RDMA offer an `explicit` interface at their lowest-levels, `software managed` abstractions like page migration can be added to make the interface more `implicit` [131]. In general, `explicit` interfaces are more general than `implicit` interfaces and can be abstracted through system software like the OS or language runtime.

In NUMA systems, the OS is responsible for choosing which NUMA domain to allocate memory from. This can be a complex decision and much effort has gone into studying such allocation policies [159].

More generally, operating systems may dynamically move pages between local and remote memory via paging. This `(implicit, software managed)` interface effectively treats local memory as a cache for disaggregated memory. This approach is taken by many projects in disaggregated memory due to the generality and ease of adoption `implicit` interfaces provide [97, 246, 104, 158, 9, 171].

Note that paging-based techniques vary in their degree of `hardware assistance` and `explicit` user control; they are not strictly in one camp or the other. Of particular note is the interface described by LegoOS, in which the authors describe a `hardware assisted` cache, and demonstrate the concept using a `software managed` technique [246]. This demonstrates the orthogonality of these dimensions. There are also variations along several of the dimensions listed in section 4.1.1.3. For example, the Mojim project uses virtual memory to present an (`isolated, durable, private`) interface to non-volatile memory (NVM) [303].

Aguilera, et al. present an mmap-style approach that adds some `explicit` properties to paging based techniques and uses filesystem semantics to handle naming [6]. Mirage and Mach's memory server added concepts of leases and explicit mapping [93, 92]. Later in this chapter, I will describe and evaluate a (`hardware assisted`, `implicit`) system I developed to accelerate the paging process.

4.1.2.5 Language-Based Approaches ($*$, `SW Managed`)

Some programming languages present a `software managed` interface to remote memory with a range of approaches along the `explicit/implicit` axis. Partitioned global address space (PGAS) languages make it appear as if some variables are `shared` while others are `private` [54, 186, 60, 247]. PGAS is partially `explicit` because users must choose which variables can be shared remotely while the language runtime `implicitly` handles data movement and caching. Actor models present another partially `explicit` interface where objects are explicitly shared through channels, but the language handles much of the complex sharing semantics described in §4.1.1.4. AIFM allows users to allocate "remoteable" objects that can be moved to disaggregated memory by the language runtime as needed [233]. The runtime manages updating pointers as objects are moved between local and remote memory.

These languages are typically implemented such that they can take advantage of `hardware assistance` when available, but fall back to software techniques when needed [41].

4.1.2.6 Data Stores (`Explicit`, `SW Managed`)

Many systems expose remote memory through higher-level software constructs like databases ([143]) and key value stores. Memcached is a widely used system for memory object caching that operates entirely in software [174]. Other systems use RDMA `hardware assistance` to accelerate access to these stores [234, 81]. Because these systems are typically `shared`, `isolated`, and (sometimes) `durable`, they must address the issues in section 4.1.1.4 and there are a wide range of approaches [292, 271, 63]. Lower-level `(explicit, hardware assisted)` interfaces may enable new abstractions as proposed by Volos, et al. [138]. Remote regions allow users to move objects between local and remote memory using file semantics [6]. Others provide a more complete file system interface [238, 160, 295].

4.1.3 Implications for Future Research

If we re-consider Figure 4.1, we observe that the extreme corners of the design space are well explored by established techniques like paging and RDMA. Much of the recent academic work has thus focused on hybrid designs that trade off between the dimensions (see Figure 4.2). For example, Infiniswap moves the `software managed` paging approach down the `HW/SW` dimension using RDMA [104]. In general, I believe that approaches that fix one dimension while exploring another represent a significant opportunity for progress. Indeed, previously impractical approaches may become practical with changes to an orthogonal dimension.

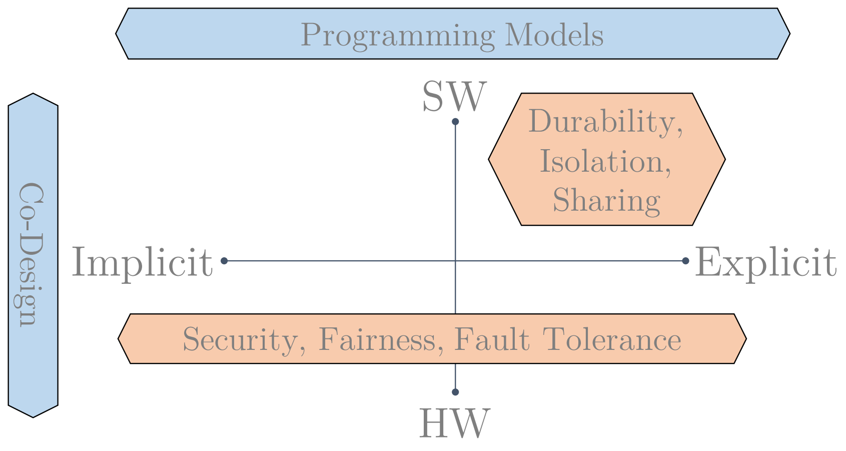

Programming Models
Co-Design
SW
Durability, Isolation, Sharing
Implicit
Explicit
Security, Fairness, Fault Tolerance
HW

Figure 4.2: Research opportunities in the disaggregated memory design space.

The logical disaggregation techniques I present in Chapter 5 can help move designs along the `Implicit/Explicit` dimension, while advances in agile hardware design like those described in Chapter 3 allow us to explore novel hybrid `HW/SW` designs.

Outside the core design space, the dimensions laid out in §4.1.1.3 and §4.1.1.4 represent opportunities for future research. Recent work on key-value stores has explored the `durable`/`ephemeral` dimension [141], while others have continued the long tradition of research into memory consistency [292]. New hardware may enable progress on issues of security, fault tolerance [53], or fairness. For example, memory blades could authenticate memory requests using capabilities or access control lists [291].

Finally, the impact of other disaggregated resources must be considered. Disaggregated accelerators and storage may lack the sophisticated software stacks required to participate in `software managed` interfaces, while full `hardware assistance` may be prohibitively expensive on legacy nodes. Interfaces must provide a flexible interface to bridge this gap and provide a path to future upgrades. I revisit these ideas in Chapter 6.

4.2 Paging-Based Approaches: The Page-Fault Accelerator

In the previous section, I presented a wide range of possible approaches to disaggregated memory. In this section, I focus on one particular strategy: demand paging to dedicated memory blades. In this approach, I use the OS's virtual memory system to treat local memory as a cache of much larger memory servers called *memory blades*. Along with my collaborators Howard Mao and Emmanuel Amaro, I used Chipyard to design an implementation of this system using Rocketcore and Linux [204].

As I presented in §4.1.2.4, paging is appealing because it is `implicit` and requires few changes to applications. However, it is not without limitations. Traditionally, paging has been backed by slow disks with access latencies in the milliseconds. This led to a fully `software managed` approach that uses sophisticated algorithms that can take several microseconds for every cache miss. With today's high performance networks, however, these algorithms begin to dominate the latency of a remote memory access.

An alternative is to have fully `hardware managed` DRAM caches [277, 151, 169]. These eliminate much of the overhead, but lack the sophistication and application-level insight of OS-based approaches. For example, operating systems often use significant memory for optimistic pre-fetching and caching of disk blocks. A hardware-managed cache may choose to store these in remote memory, while the OS would simply delete them.

These two approaches exemplify the trade-offs along the hardware/software axis of the disaggregation design space. However, this design space is not a strict dichotomy. In the coming sections, I describe a design called the *page fault accelerator (PFA)*, that compromises between hardware and software management. The sophisticated paging algorithms continue to be managed in software, but the primary page fetching behaviors occur in hardware. §4.2.1 describes remote memory paging in more detail, including its limitations. I then present my hardware-accelerated solution in §4.2.2. In §4.2.5, I show that this technique can improve page fetch latency by 2.2x and end-to-end runtime by 20 %.

4.2.1 Remote Paging Background

Many architectures expose the abstraction of virtual memory. While the implementation of virtual memory is fairly similar across architectures and operating systems, for concreteness I will use Linux running on the RISC-V ISA for most examples in this chapter[1].

Figure 4.3 shows a typical flow for translating a virtual address. Most translations occur completely in hardware and require no immediate OS intervention. However, when physical memory is constrained, the operating system may choose to store logical pages in secondary storage (called *paging*). In this case, some mappings are *invalid* and not present in physical memory. Accessing these pages requires immediate OS intervention to resolve. Throughout this chapter, I will refer to the logical data as a *page*, and the physical location in memory as the *page-frame* or simply *frame*.

There are three main contributors to page fault time: trap time, processing, and backing store access. Trap time depends on the microarchitecture of the underlying system. On an Intel Haswell CPU, this takes approximately 800 ns. Rocketcore is a much simpler in-order processor, resulting in very fast traps of about 100 ns. Backing store access is dependent on the device technology. Spinning hard disks will have access times in the tens of milliseconds, SSDs are closer to one or two milliseconds, while Infiniband will have a page read latency of about two microseconds.

[1]RISC-V privileged architecture version 1.10 [285] and Linux 4.15 [269]

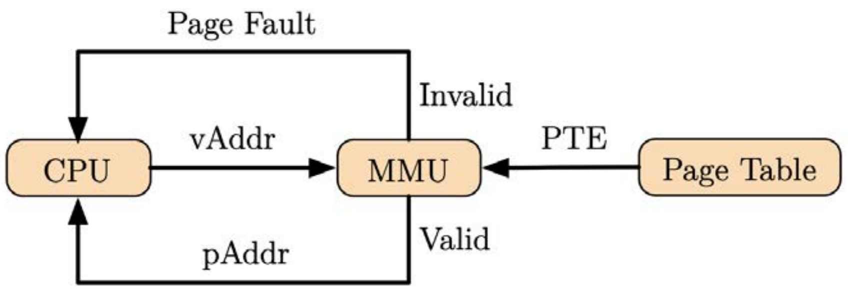


Figure 4.3: Flow chart for virtual to physical address look up in a typical virtual memory system. The translation look aside buffer (TLB) caches translations. The page-table walker fetches mappings from main memory when the TLB misses. Most mappings are valid and can be returned directly to the CPU, but invalid mappings result in a trap to the OS.

The remaining component is processing time. This is the time spent by the operating system running its paging algorithms and looking up metadata. I collectively refer to these tasks as *bookkeeping* (see §4.2.4.1 for details of this process). These steps can be time consuming and degrade the performance of other parts of the system. On the microarchitectural side, we see a large number of complex data structures being accessed. These can require many small memory accesses and tend to fill up caches with rarely accessed data. The trap itself may also disrupt the CPU pipeline, branch prediction, and other components. From a software perspective, many of these operations require locking and synchronization. This can add lock contention and disrupt the memory system of other cores, not to mention the additional TLB flushes. This process takes approximately 5 µs on Rocketchip when there are no competing workloads. The impact on microarchitectural state is more difficult to quantify. I describe the paging algorithms in detail in §4.2.4.1 and measure their performance cost, including their impact on microarchitectural state, in §4.2.5.2.

Taken together, these overheads mean that a naïve paging-based approach will see significant slowdowns, even with an infinitely fast network. To demonstrate this, I modified the Linux kernel to use reserved physical memory as a swap device. This means that our backing store is effectively a single `memcpy`. Figure 4.4 plots several benchmarks' runtime as they are run under increasingly memory constrained environments. Note that even without waiting for secondary storage, applications can slow down by as much as 12.5x due to paging overheads.

4.2.2 The Page Fault Accelerator

Much of the work done during a page fault does not need to occur for the application thread to make progress. Allocating free frames or updating page metadata can be performed at any time. Other tasks may be more efficient in hardware than in the OS; the walking of page-

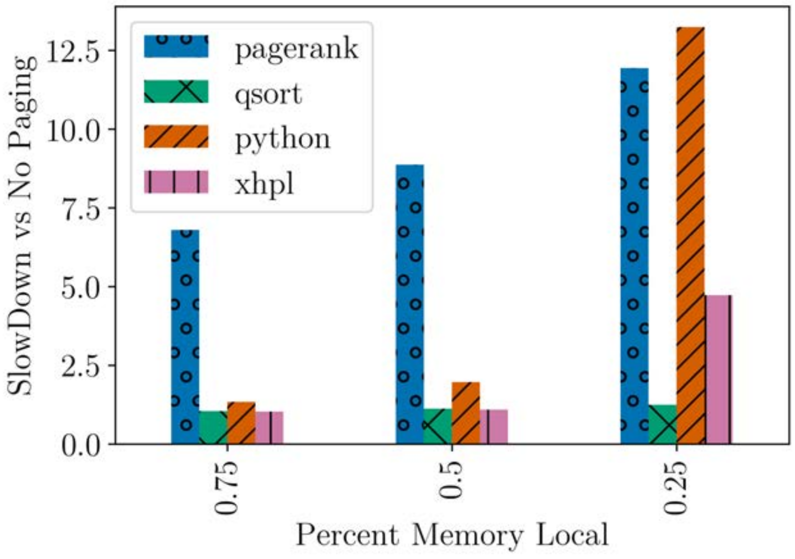


Figure 4.4: Application slow-down when paging to local memory. I first measure the peak memory requirements for each benchmark. I then reduce the amount of available local memory as a fraction of peak.

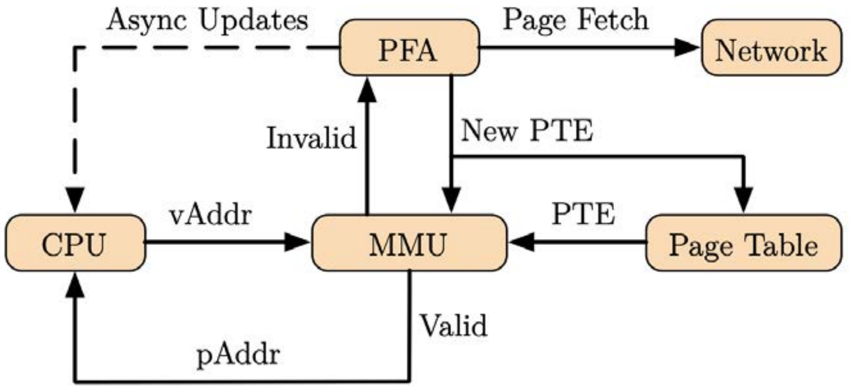


Figure 4.5: Paging with the PFA. Instead of an invalid PTE causing a trap to the OS (as in Figure 4.3), invalid pages are passed to the PFA to be fetched from remote memory. The PFA may still cause a trap if it cannot handle the request (e.g., full queues). The OS can then asynchronously query the PFA for a list of fetched pages.

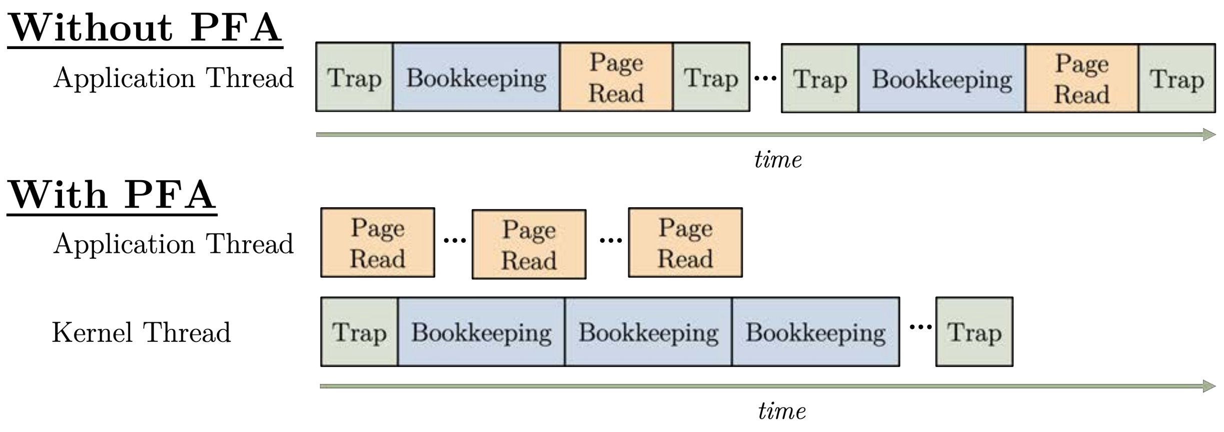


Figure 4.6: Timeline of page-fault processing with and without the PFA. Without the PFA, the OS must be invoked on every page miss. The PFA allows this bookkeeping to occur any time after the fetch in a separate kernel thread. Only the actual page read must occur before the application can be restarted.

tables for example. I propose a hardware accelerator that performs only the bare-minimum of copying a remote page into a pre-allocated frame, updating the relevant page table entry (PTE), and restarting the application. Figure 4.5 shows the new page fetch process.

While this does not eliminate the need for software management of page meta-data, it does provide considerable flexibility to the OS in how such tasks get scheduled. Figure 4.6 illustrates the difference from the perspective of the OS. One immediate benefit is that the OS can schedule this bookkeeping task when the system is idle or the application thread is blocked. Another benefit is that bookkeeping tasks can now be batched. Batching improves cache locality and amortizes the microarchitectural impacts.

The primary interface to the PFA is through a number of memory-mapped queues: FreeQ, NewQ, and EvictQ. The FreeQ contains unused page frames that the PFA can use for fetching new pages, the NewQ reports any recently fetched pages to the OS bookkeeping thread, and the EvictQ contains a list of local pages that should be stored in remote memory. Remote page metadata is stored in the page table using a custom PTE format.

4.2.2.1 Remote Page Table Entry

The MMU must be able to recognize that a page is remote and the PFA must know where in remote memory that page is. I do this through a special PTE format. Fortunately, the RISC-V ISA leaves most PTE fields undefined for entries with the valid bit cleared. I exploit this by marking remote pages as invalid and defining a new PTE format for these invalid entries. Figure 4.7 depicts this new format. The fields are as follows:

- **PageID**: This acts as an address in remote memory for the remote page. It is used by the PFA to look up pages in remote memory, and by the OS to identify each page

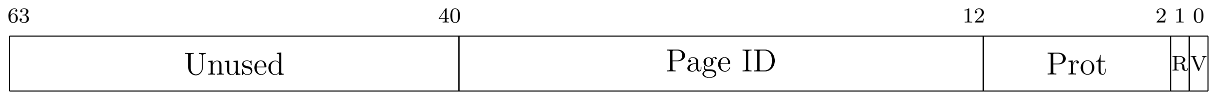


Figure 4.7: Remote PTE Format. The **Page ID** is a unique identifier of this page and serves as a remote memory address. The **Prot** field contains the permission and metadata bits that should be set after a page is fetched. The **R** bit indicates that this page is remote while the **V** bit indicates that the PTE is not a valid mapping.

during bookkeeping.

- **Prot**: This sets the protection bits that the PFA will use when fetching a page. These bits include things like read/write permissions, as well as other page metadata as defined by the RISC-V specification.
- **R**: This bit indicates that a page is remote when the valid bit is clear.
- **V**: This indicates whether a page is valid. A valid page is currently in main memory and would not trigger a page-fault. This is also referred to as the *present bit* in Linux.

An interesting feature of this design is the use of pre-defined protection bits. This includes a valid bit which can be cleared by the OS before evicting to trigger a page fault on this page immediately after fetching (a useful debugging feature). Also, bits 8 and 9 are reserved for software by the RISC-V ISA and can aid the OS in bookkeeping and debugging.

4.2.2.2 PFA Operation

There are three core components of PFA operation. The first is page eviction where local pages are written to remote memory. Next is page fetch where the PFA synchronously moves a remote page into local memory. Finally, the OS must asynchronously manage the PFA by allocating physical memory for new pages, and checking for fetch notifications.

Eviction The PFA handles all communication with the memory blade, including page eviction. The basic procedure is as follows (see Figure 4.8):

1. The OS identifies pages that should be stored remotely.
2. It evicts them explicitly by writing to the EvictQ.
3. The PFA sends a remote memory write command to the NIC which reads the page and sends it to remote memory.
4. When the send is complete, the PFA updates the EvictQ status to notify the OS.

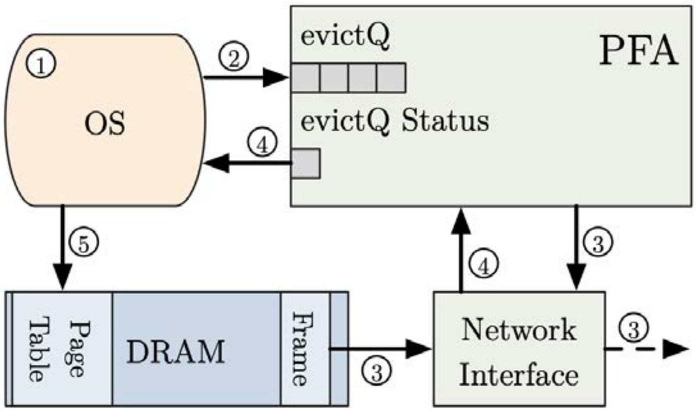


Figure 4.8: Detailed eviction flow

5. The OS stores a page identifier in the PTE and marks it as remote once the PFA eviction is complete.

In addition to the three main queues, there are a number of other maintenance registers that are used for querying queue status and initializing the PFA. I will mention one status register here; the `EVICT_STAT` register. When a page is placed on the evict queue, the PFA begins transferring it to remote memory, but does not block the OS. This allows the OS to perform useful work while the eviction is taking place, potentially hiding some of the write latency. To re-use the page frame, however, the OS must poll the `EVICT_STAT` register to ensure the write has completed.

Fetch The primary function of the PFA is to automatically fetch pages from remote memory when an application tries to access them. It does this by detecting page table entries that are marked remote and transparently re-mapping them to the next available free frame. The basic operation is as follows (see Figure 4.9):

1. Application code issues a load/store for a remote page.
2. The MMU detects a remote page using the valid and remote bits and requests it from the PFA.
3. The PFA issues a remote memory read command to the NIC, providing the next available frame from the FreeQ.
4. The PFA updates the PTE with the new physical address, clears the remote bit, and writes it back to the page table.
5. The PFA pushes the virtual address of the fetched page to the NewQ.

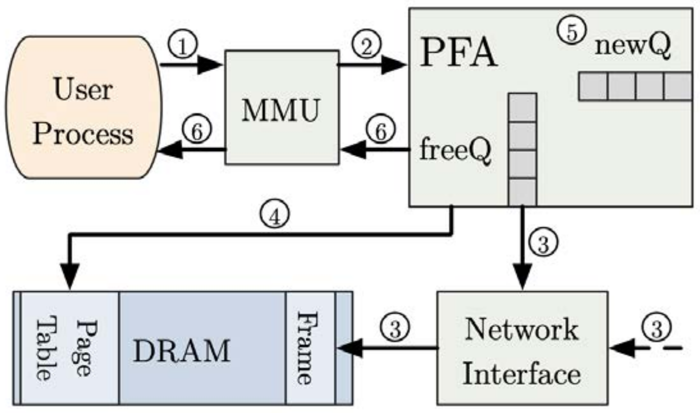


Figure 4.9: Detailed fetch flow

6. The MMU updates the PTE and restarts the application.

Metadata Management The OS should ensure that there are sufficient free frames in the FreeQ to ensure smooth operation. If a remote page is requested and there are no free frames, the PFA will trap to the OS with a conventional page-fault. The OS must enqueue one or more free frames before returning from the interrupt. This may involve evicting pages synchronously in the page-fault handler. Similarly, the OS needs to drain the new page queue periodically to ensure it does not overflow. This will also trap to the OS with a conventional page fault if full.

4.2.3 Hardware Implementation

There are many possible arrangements for physically disaggregated hardware. For this project, I assume an approach based on the FireBox system I introduced in Chapter 2. In this arrangement, shown in Figure 4.10, compute elements are packaged into self-contained systems in package (SiPs) with a modest amount of on-package high-bandwidth memory. The bulk of system memory is contained on dedicated *memory blades* that include a large amount of DRAM and a high performance RDMA-like network interface.

Along with my collaborators Howard Mao and Emmanuel Amaro, I used Chipyard to design an implementation of this system. Table 4.11 describes the hardware parameters I used in my evaluation. The compute SiPs used Rocket core as the compute element. Networking was provided by an RDMA-capable NIC, coupled with a parameterized network model configured to resemble Infiniband's 2 µs round trip latency and 200 Gbit/s throughput. I designed the software for this SiP using FireMarshal and evaluated its performance using FireSim. Howard Mao describes the hardware implementation of the memory blade in detail in his dissertation [169].

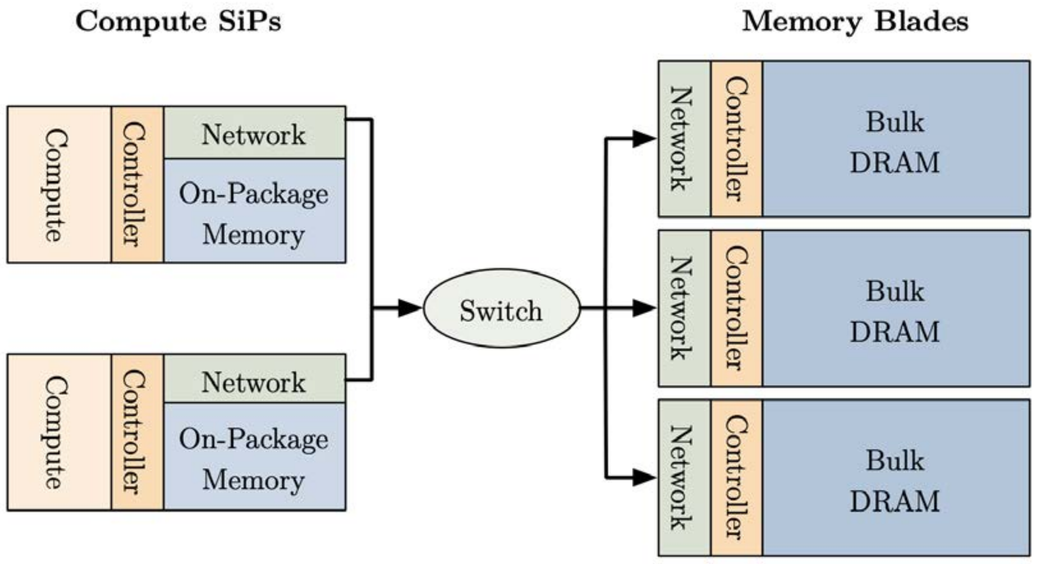


Figure 4.10: Example system with physically disaggregated memory. Compute SiPs contain the primary programmable computational resources while memory blades contain the bare minimum hardware to serve memory requests. All nodes contain some amount of controller logic to handle protocol processing and system administration.

CPU Type	Rocket (5-stage in order)
CPU Frequency	3.2 GHz
Caches	16 kB Data and Instruction
NW Topology	Single Switch
NW Bandwidth	200 Gbit/s
NW Link Latency	2 µs
Remote Page Read	4.8 µs
Remote Page Write	4.8 µs

Figure 4.11: System parameters used for evaluation.

While I present the completed product here, it is worth noting that this hardware was built concurrently with the Chipyard project. Many of the methodologies introduced in Chapter 3 were developed in response to challenges we faced in this memory disaggregation project.

4.2.3.1 Functional Model

Following the methodology of Chapter 3, I first implemented a golden model of the PFA in Spike. Due to its simplicity, the PFA implementation required only a few weeks of implementation effort and less than 1000 lines of code. With Spike, software development was able to proceed concurrently with the concrete hardware design. Furthermore, unit tests developed under Spike were used to validate the hardware implementation, reducing debugging effort. In all, the only software change that was needed to go from Spike to a concrete implementation in FireSim was one extra TLB flush due to a difference in TLB design between Spike and Rocketchip.

4.2.4 Linux Integration

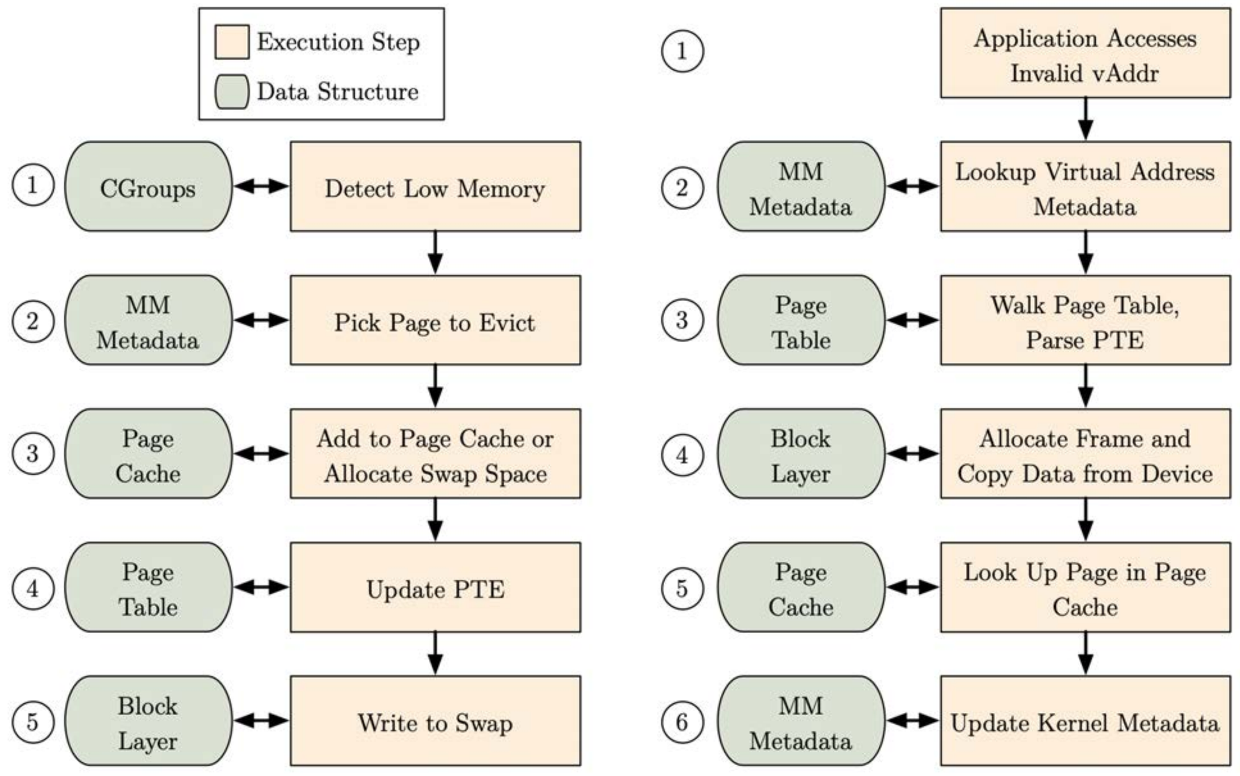
I modified the Linux kernel to support the PFA. The majority of software development was done using the functional simulator.

4.2.4.1 Non-PFA Paging in Linux

I now briefly describe paging in vanilla Linux. Note that the kernel internally uses the term *swap* in reference to all paging activity, I use these terms interchangeably. For a more complete discussion of memory management in Linux, see [43]. Figures 4.12a and 4.12b show the steps involved in evicting and fetching pages, respectively.

Page Reclaiming Linux manages memory limits on a per-task basis. A task refers to the kernel-specific abstraction of a process. Each task has its own resource limits which are exposed to system administrators through the control group (cgroup) interface. When a task approaches its assigned limit of a certain resource, it is throttled in a resource-specific manner. In the case of memory, the kernel attempts to free task-assigned memory. It will first attempt to shrink any file caches, especially clean disk blocks that can simply be deleted without requiring any disk activity. If shrinking caches is not enough, the kernel begins to page non-file backed pages called *anonymous pages*. This is done using a pseudo-least recently used (LRU) eviction algorithm.

Page Eviction Figure 4.12a shows the steps involved in evicting pages to a swap device in Linux. Paging was originally intended to use hard disks as the backing store, and this is reflected in the design of paging in Linux. To swap, one or more block devices must be formatted and mounted as swap devices. Linux then uses the block offset on this disk as a unique identifier for an evicted page. To support more complex paging schemes such as page compression or heterogeneous memory, Linux introduced the transcendent memory (TMem) layer [165]. This scheme still uses disk offsets as identifiers, but completely bypasses the block layer. This is important because many optimizations in the block layer like write coalescing and block reordering are not suitable for these alternative paging devices. Evictions do not

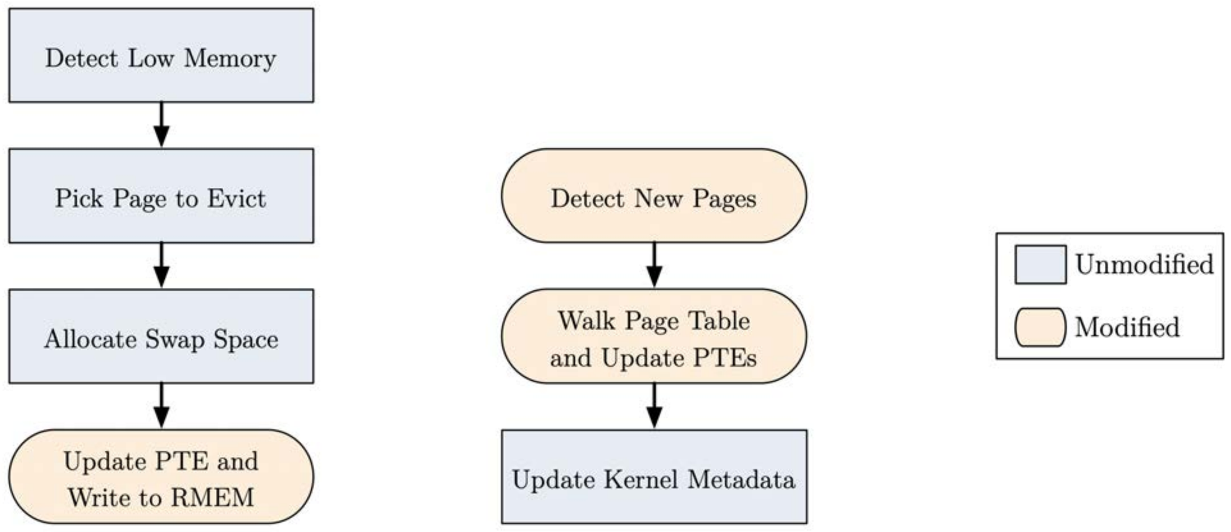


(a) Page Eviction (b) Page Fetch

Figure 4.12: Steps taken by Linux to evict or fetch a page from a swap device.

immediately result in writes to TMem or a swap device. Instead, pages are stored in a data structure called the page cache. The page cache is used for all block device I/O, not just swapped pages. This page cache helps reference count shared pages, and hedges against poor eviction choices. Once a page is no longer physically available, Linux replaces the corresponding PTE with a swap entry which clears the valid bit, and uses the remaining bits to store the swap device ID (called *type* in the kernel) and block ID (called *offset*). When changing PTEs, RISC-V requires the OS to flush the translation look-aside buffer (TLB). This forces a page-table walk on the next access to this virtual address. Finally, the kernel begins a write to the swap device in the background.

Page Fetch Figure 4.12b shows the steps taken when a user program attempts to access a page that has been swapped out. The MMU first notices the invalid PTE and issues a page fault to the OS. Note that hardware does not examine the remaining bits; the swap entry is purely a software construct. Upon receiving a page fault, Linux first determines if the requested virtual address has been assigned to this task. It does this by iterating through regions of virtual memory called virtual memory areas (VMAs). VMAs are groups



(a) Page Eviction Changes (b) Page Fetch Changes

Figure 4.13: Major changes to Linux paging to accommodate the PFA

of virtually contiguous pages that all share properties like permissions and file mappings. They allow the kernel to reduce the amount of metadata it must store, and batch metadata changes. If a VMA is found for the faulting address, the OS begins a page table walk to locate the corresponding PTE. There are several reasons that a page fault may occur, the OS must check the PTE to determine the cause. Assuming the cause was an invalid PTE, the OS then searches the page cache for this page. This is in case some other process that shares it has already brought it in. If the page is not found, a new frame is allocated and a transfer is initiated to read the page from the swap device. If the page is found in TMem, the transfer occurs synchronously, otherwise the process initiates the transfer and yields to the scheduler, resulting in a context switch. When the transfer is complete, the kernel changes the PTE from a swap entry to a valid PTE with permissions defined by the VMA. Finally, the kernel updates page tracking metadata. This includes the LRU lists maintained by the eviction algorithm, VMA membership, and a number of other kernel subsystems. Note that several of these updates require synchronization with other kernel threads. Once all bookkeeping is complete, and the PTE is updated, the kernel flushes the TLB and restarts the application.

4.2.4.2 PFA Modifications

The PFA handles steps 1, 3, and 4 from figure 4.12b synchronously in hardware while the Linux kernel manages steps 2, 5, and 6 asynchronously. This changes a number of assumptions underlying baseline paging behavior. Figure 4.13 summarizes these changes.

Fetched Page Permissions Linux uses the faulting virtual address to make a number of decisions during the page fetch process. For instance, the permission bits are taken from the VMA. With the PFA, however, the OS must decide on this information at *eviction* time. Pre-allocating physical frames is not an issue in my system because frame selection does not depend on the VMA in non-NUMA systems. Permission bit selection is more problematic. My approach is to encode page permissions in the remote PTE at eviction time. These will be used provisionally when the PFA first fetches the page. I then update those permissions while performing bookkeeping. In practice, this is unlikely to cause problems as permissions rarely change. Furthermore, Linux is able to correct inappropriately restrictive permissions during page-faults. However, there may be security concerns if permissions are made more restrictive while a page is remote. This vulnerability exists in the window between page fetch and bookkeeping. To mitigate this concern, the OS would need to be modified to update remote PTEs when changing VMA permissions.

Asynchronous Bookkeeping In normal paging, Linux is able to update metadata as soon as a page is fetched. With the PFA, this bookkeeping is delayed for a bounded but potentially non-trivial period of time. Many of these bookkeeping tasks are in support of heuristics or resource accounting. Delaying these tasks reduces the accuracy of various algorithms, but does not result in incorrect behavior. Others are needed for correct execution (e.g., VMA membership or shared page tracking for copy-on-write). I address these correctness issues by performing bookkeeping preemptively before any of the related algorithms execute. These tasks may be fairly common, but they are unlikely to actually involve a recently fetched page. To avoid preemptively performing bookkeeping, I use one of the reserved bits in the PTE protection field to indicate a page that has been recently fetched but not yet processed. This bit gets set at eviction time, but is cleared during bookkeeping. I always refill the FreeQ after bookkeeping.

Swap Device and Block ID Allocation Linux assumes that all swap activity is backed by a block device and it uses the physical address on this device to identify all evicted pages. This block ID is needed during the bookkeeping process to identify the page. To address this problem I make a number of simplifying assumptions.

1. **A real swap device is available.** Even if it is not used, a swap device must always be available. I use a ram-based file system (ramfs) to trick the kernel into thinking it has a large disk attached. Ramfs only consumes physical memory when written to, even though it appears to the kernel as a large disk.

2. **There is only one swap device.** This allows me to not track the device ID. This is achieved by making the ramfs sufficiently large to address all swap activity.

3. **Block IDs are contiguous on the integers $(\mathbf{0}, \mathbf{2^{28}}]$.** This allows the block ID to be packed into the remote PTE format. I achieve this by ensuring that the ramfs is the

same size as the memory blade (and less than 2^{28} pages). Since block IDs correspond to physical offsets on the swap device, the system is guaranteed to never see an invalid block ID.

While these assumptions hold, I am able to compress the swap entry into a 28 bit PageID by eliding the type, and using the offset directly. Finally, I avoid overheads in the block layer by implementing the PFA as a TMem device. Since bookkeeping is asynchronous, and eviction occurs earlier in the process, this TMem plugin simply returns immediately. The current implementation evicts synchronously. This is because the expected write time is much smaller than a scheduling quantum and asynchronous eviction would result in wasteful context switches. Future implementations may attempt to overlap eviction with low-latency tasks such as bookkeeping.

Baseline Swapping I modified Linux to use the remote memory blade directly while paging rather than using the PFA. This was done by implementing a software interface to the remote memory blade as a TMem device. The swapping mechanism uses a custom network driver that provides zero-copy semantics and bypasses the normal Linux networking stack.

4.2.5 Evaluation

4.2.5.1 Experimental Design

My evaluation is based on two benchmarks with significantly different access patterns. The first is quicksort (*Qsort*). This benchmark first allocates a large array of random numbers, and then sorts it using the well-known quicksort algorithm. Quicksort is a divide and conquer algorithm that automatically partitions the input array into small local blocks before performing a final sort. This leads to excellent cache behavior and predictable access patterns. This benchmark performs no file I/O and is never blocked on OS interactions.

The other benchmark is a de-novo genome assembly benchmark (*Gen*). Gen begins by loading a large text file that represents raw genome data. Raw genome data consists of short overlapping sequences of base-pairs called *contigs*, the goal is to align these overlapping contigs into a single contiguous sequence representing a genome. This is done by loading contigs into a large hash table and probing into it repeatedly to find matching sequences. This leads to very little locality and unpredictable access patterns. Furthermore, Gen performs file I/O on the input, which allows for more complex OS interactions.

4.2.5.2 End-to-End Performance

I ran the benchmarks under a cgroup in Linux to reduce the available memory and emulate a system where applications would need to share limited local memory. This is the same

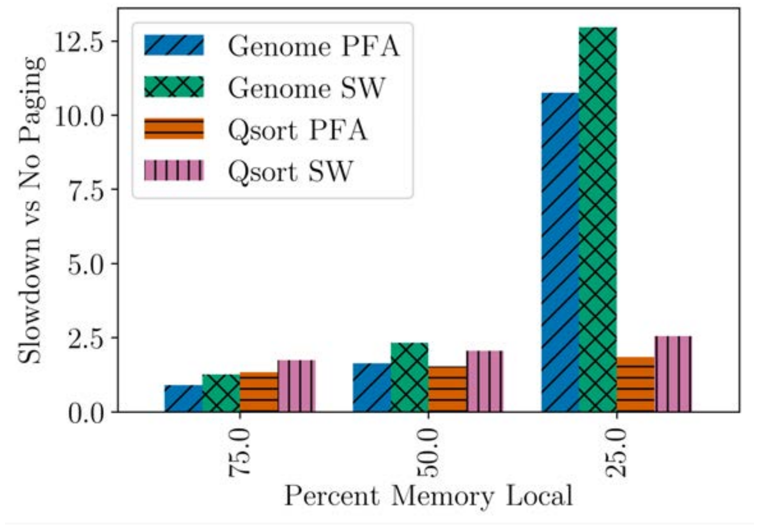


Figure 4.14: PFA vs Baseline without kswapd. Applications run approximately 20-40% faster when the PFA is enabled.

mechanism that system administrators use today to control application memory consumption. I configured the PFA to allow up to 64 outstanding page faults before bookkeeping must be performed.

Both applications use 64 MB of memory at their peak. I then varied the cgroup memory limit from 100% (64 MB) down to 25% (16 MB), triggering increasing levels of paging. For both benchmarks, the PFA reduces end to end run time by up to 20 %. I now analyze the sources of this performance improvement.

Fetch Times I begin my analysis by looking at the key metric of average fetch time. This is the time between when an application attempts to access a remote page, and when it is able to continue processing. In this experiment, I use a simplified memory blade and network implementation with a constant 4 µs access latency in order to better understand local overheads. Figure 4.15 plots the time for accessing a single remote page on an unloaded system. I classify time into four categories:

- **Trap:** The time for the hardware to detect an invalid access and context switch to the OS.
- **Proc:** The time spent processing the page locally.
- **NIC:** The time spent interacting with the NIC.
- **MemBlade:** The time spent on the network and in the memory blade.

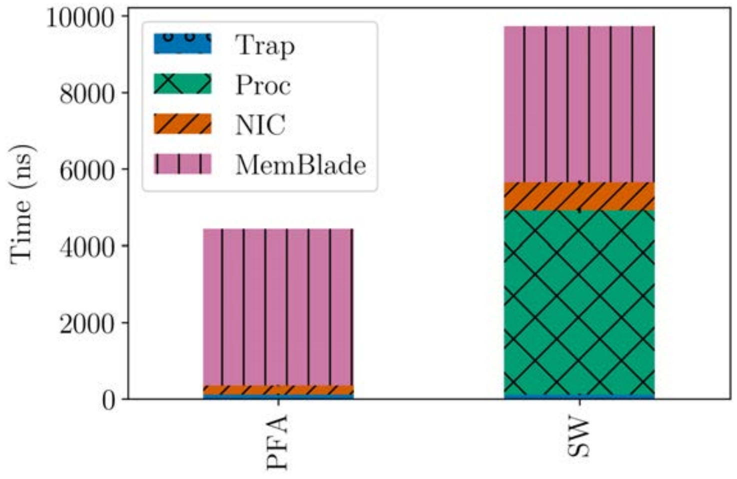


Figure 4.15: Breakdown of time in fetching a single remote page. All data are the average of 10 runs. Error bars represent standard deviation (but are almost too small to be seen). Note that local processing time, including the trap and NIC interaction, only accounts for 8% of time with the PFA, but accounts for over 50% of time for the baseline.

Note that the trap overhead is a very small fraction of total time (just 113 ns). This is a result of using a simple in-order RISC core like Rocket. This overhead is more significant on more complex architectures like those found in servers and mobile devices. Next, note that the time spent on the network and in the memory blade accounts for less than half of the time in the baseline implementation, but completely dominates the PFA fetch time. This effect will be even more pronounced as network and memory blade performance improves. For example, if **MemBlade** time were reduced to 1129 ns to simulate a 1 Tbit/s link with 1 µs round-trip latency as predicted in Chapter 2 for photonic networks, then client-side processing would account for 83% of time in the baseline but only 23% of time with the PFA (Figure 4.16). Finally, note that the **NIC** time in software is larger than with the PFA. This is due to a more efficient hardware to hardware interface between the PFA and the NIC. While not visible in the figure, the actual PFA-specific processing takes only 1 cycle in hardware, the remaining time is split between detecting and delivering the remote PTE to the PFA (**Trap**), and interacting with the NIC (**NIC**). The total time to fetch a page with the PFA is 2.2 times faster than the baseline, but this does not tell the whole story. The PFA does not eliminate the work that is done during **Proc**, it simply moves it to another thread. Likewise, the 113 ns trap overhead may seem small, but this does not account for the microarchitectural effects of the handler on application performance when it restarts.

Total Page Faults One key function of the PFA is to reduce the number of page faults due to paging. Recall from §4.2.1 that there are many causes for faults, in Figure 4.17 I plot the number of paging-related faults each benchmark experiences as a fraction of total faults. The

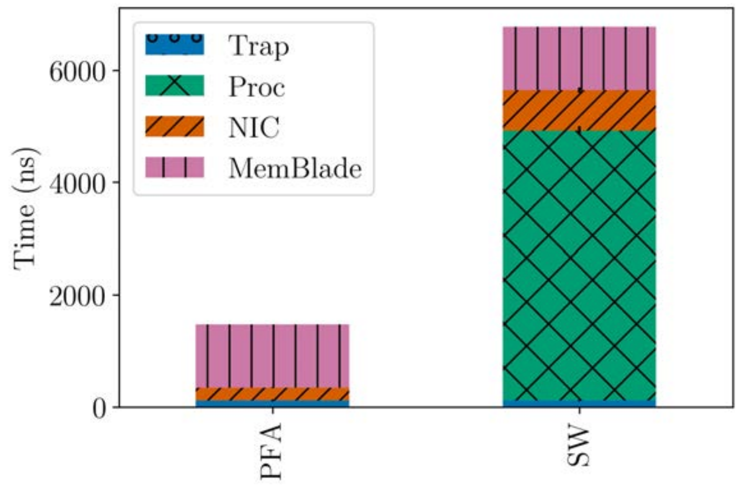


Figure 4.16: Breakdown of time in fetching a single remote page from a hypothetical fast memory blade with 1 µs page read latency. As network and memory technology improves, the relative benefit of the PFA increases (from 2.2x faster with the baseline memory blade to 4.6x faster with the optimistic memory blade).

first thing to note is that the number paging-related faults decreases by approximately 64 when the PFA is used. This is because the PFA interrupts the OS to perform bookkeeping only when its queues are full (every 64 fetches in this experiment). However, these only account for 45% of faults, even in the worst-case of of Qsort with 25% local memory. The more complex Gen benchmark has an even lower fraction of paging-related faults. While there are certainly some savings due to fewer kernel crossings, they are not frequent enough nor long enough to explain all the performance benefits we see end-to-end.

Bookkeeping Time While the PFA does reduce the number of paging-related faults, the kernel still needs to perform bookkeeping on the same number of pages. This batching means that more work is performed per page fault with the PFA. Figure 4.18 shows total time spent bookkeeping, regardless of the number of page faults. What we see is that while the number of evicted pages is the same in both configurations, using the PFA leads to a 2.5x reduction in bookkeeping time on average. The same code path is executed for each new page, but the PFA batches these events. This leads to improved cache locality for the OS, and fewer cache-polluting page-faults for the application. The result is that, even in the worst case, the PFA spends less than half its time handling paging-related faults while the baseline spends about 80%.

Scaling Figure 4.19 shows the improvement in end-to-end runtime due to the PFA. While the improvement is significant, the savings are constant. This is because the PFA does not

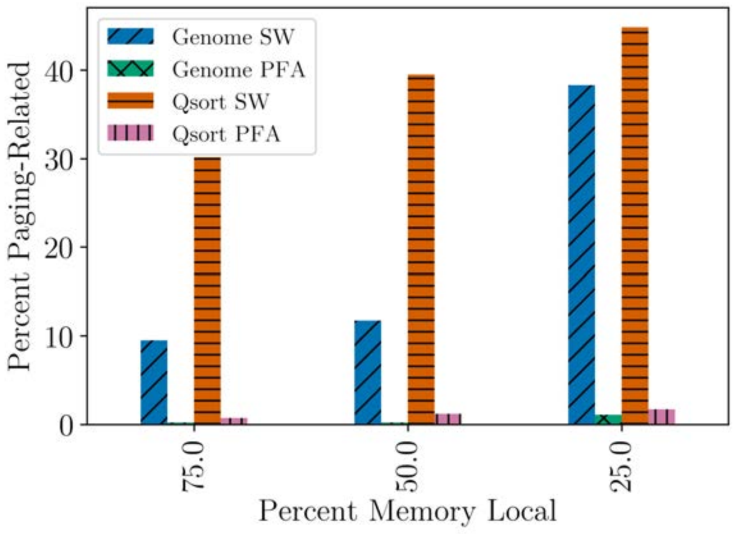


Figure 4.17: Number of paging-related faults as a fraction of total faults experienced.

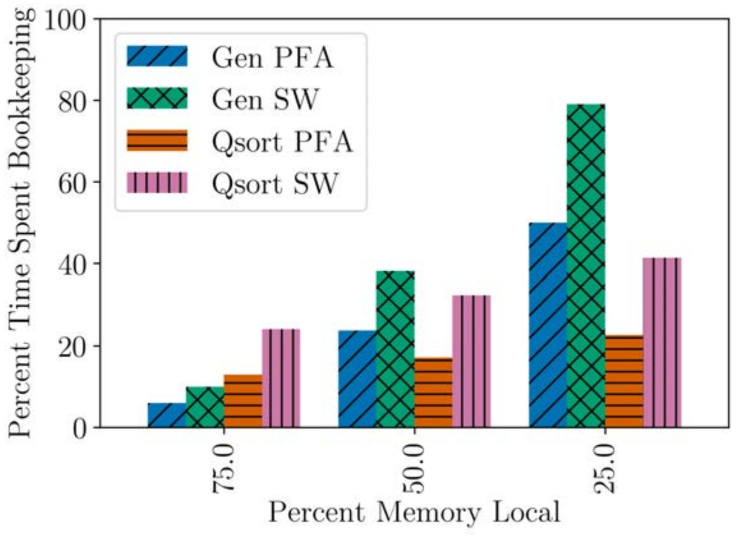


Figure 4.18: Proportion of time spent bookkeeping.

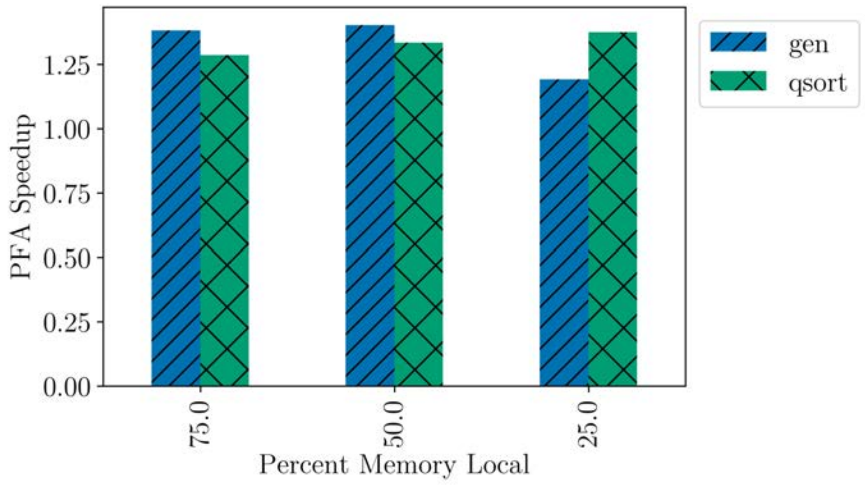


Figure 4.19: Total runtime improvement due to the PFA

change any of the caching algorithms, and therefore experiences the same number of faults.

4.2.5.3 Evaluation Limitations

This evaluation used a single, in-order, CPU core and relatively simple benchmarks. Both Gen and Qsort operated over a fixed dataset for a fixed period of time. This predictability meant that they made effective use of a known amount of memory. Many real-world applications are less predictable. Microsoft reports that many applications do not use a significant fraction of their memory after first use [153]. For the median VM, this "frigid" memory accounts for 50 % of all memory used. For these applications, we would expect remote memory paging to have a much smaller impact on end-to-end performance. The impact of paging overheads may also be reduced in applications that use large page sizes to amortize virtual memory management costs. While these effects make remote memory paging more appealing, they do not reduce the cost of handling the pages that *do* fault. In this evaluation, I have shown how the PFA can improve the performance of a single fault, regardless of the application that caused it. I have also shown how bookkeeping costs scale with paging activity when it does occur. For any workload using software-only remote paging, we can expect to similarly see a 2.2x reduction in individual page fault latency by using the PFA (Figure 4.15). Similarly, we can expect bookkeeping times to reduce by roughly 30 % due to batching in the background thread, but still consume a large fraction of total processing time when paging activity is frequent (Figure 4.18).

From a hardware perspective, our simple in-order core may not be representative of high performance server-grade CPUs. Rocket has fast traps because it is in-order and has a short pipeline while more complex out-of-order cores can expect much higher trap overheads. While Rocket's small caches will be more impacted by page fault handling than larger cores, its simple microarchitecture will see less disruption from the context switch. Interestingly, these differences result in very similar page fault latencies between the two platforms. While Figure 4.15 reports roughly 5 µs for trap and processing time on Rocket, my experiments on an Intel Xeon core also show 5 µs trap and processing latency.

4.2.6 PFA Conclusions

The PFA improved the performance of remote memory paging by introducing new hardware to accelerate key phases of the process. Users did not need to change their applications to take advantage of disaggregation and in fact did not need to know it was happening. However, the PFA did not fundamentally change the underlying approach to disaggregated memory. This means that applications like Gen that are not particularly cache-friendly can see significant slowdowns in a disaggregated environment, even with the PFA. The PFA pushes the boundaries of what is possible with cache-like interfaces, but it cannot change their fundamental limitations. Applications like Gen will need deeper changes to be viable on a disaggregated system. In the next section, I describe one system I designed that requires users to adapt their applications to disaggregation, but reaps large benefits from this more explicit approach.

4.3 Example Application: Process Checkpointing

So far I have focused on *how* memory disaggregation might be achieved. In this section, I present one application of this technique to improve failure recovery.

Warehouse-scale computers allow users to scale their applications to thousands of nodes. This scale enables high performance applications spanning deep learning training to massive scientific simulations. Scale also means that failures become common. For long-running large-scale applications, the chances of at least one component failing approaches certainty [224, 236]. The result is that fault-tolerance must be a first-class concern in the design of such applications. There are many solutions to this problem, of particular interest for disaggregation is the use of state checkpointing. This strategy involves periodically saving the state of a running task to some remote location in a different fault domain. If a failure occurs, the task is restarted using the checkpointed data.

One way to checkpoint tasks is to replicate the entire OS process. This requires little or no changes to the application and can be implemented using generic tools like Berkeley Lab Checkpoint Restart (BLCR), checkpoint/restart in userspace (CRIU), or Condor [83, 70, 35]. As noted earlier, `implicit` approaches like this tend to introduce inefficiencies because they cannot exploit application-level insights. In this case, process checkpoint/restart must checkpoint the entire application, even temporary or unimportant data. They must also quiesce operating system state and ensure that all user-visible state is saved properly. To avoid this, some applications take a fully `explicit` approach by manually serializing and storing all critical state on each checkpoint. As expected from an explicit approach, this requires

`neph_cfg_[create/destroy]`	Initialize the system and recover memory if needed
`neph_[alloc/free]`	Allocate a region of recoverable memory
`neph_txn_[start/commit]`	Mark a point of consistency in the program
`neph_[set/get]_usr_data`	Register a pointer to your state

Table 4.1: Nephele API

significant effort from application developers and can be tricky to make fast. Serialization is already a major bottleneck in distributed applications and it can be difficult to identify all critical state accurately. MODC is one approach that couples a task-based programming model with explicit, named, objects that are persisted to remote memory [138]. OpenFAM provides an application programming interface (API) for storing and naming persistent objects in remote memory [137].

Together with João Carreira and Howard Mao, I designed a system called Nephele that compromises between the two extremes of `implicit` and `explicit`. It is explicit because it requires users to identify memory allocations that must be checkpointed as well as safe points in their application where a checkpoint can be taken. The system then implicitly tracks changes to the critical state and replicates data as needed at these safe points. Unlike full process checkpointing, Nephele checkpoints only critical state and does not require fully transparent OS state management. However, it also does not require the use of special data structures and serialization code from applications.

4.3.1 The Nephele Interface

The Nephele interface is designed around two features: a recoverable memory allocator, and commit points. The recoverable memory allocator has the same API and semantics as the standard `malloc` function, but additionally marks memory as recoverable. As a drop-in replacement for `malloc`, the recoverable allocation API is relatively easy for application developers to integrate into existing code. Commit points allow users to mark points in their application where they are able to recover using only recoverable memory. Nephele will checkpoint all recoverable memory at each commit point. Upon restart, Nephele loads all recoverable memory into their original virtual addresses using the `mmap` system call and begins the process from its normal entry point. Memory allocation is done early in process creation to ensure that the recoverable virtual addresses are not allocated to any other objects. Users may also require additional metadata to be able to interpret their recoverable memory. Nephele provides this through a user-defined `user_data` data structure. Nephele will provide a pointer to this data structure upon restart. Table 4.1 lists the entire Nephele API.

While this API is designed to be minimally intrusive, it does require some application support. Applications must include recovery code to prepare for execution given the recoverable state. Nephele also does not manage external state like file pointers or network sockets. While not as easy as full process checkpointing, these tasks are less complex than full state deserialization and recovery. Since recoverable memory is placed in identical virtual addresses, there is no need for application-specific serialization routines.

4.3.2 Implementation

Nephele consists of a high-level layer that is responsible for tracking recoverable memory and commit points, and a low-level atomic replication library.

4.3.2.1 Atomic Replication Library

Nepehele uses an atomic replication library to manage remote memory. Clients can allocate remote memory on a byte granularity, though Nephele only uses page-sized blocks. Once allocated, users can read and write remote memory. Writes occur atomically through the use of a redo log on the server. We use an Infiniband backend to implement this library. Reads and writes use one-sided puts and gets. The client initiates an atomic commit using a two-sided RPC to the memory server which applies the redo log.

Infiniband requires users to pin virtual addresses in memory and register each region with the Infiniband driver. In this project, we used an early prototype of the replication library that managed pinned memory at a fine granularity, leading to significant performance overheads. Future versions of the library used a more sophisticated memory management algorithm that eliminated many of these constant overheads. The replication library was designed by João Carreira and is described in more detail in [55].

4.3.2.2 The Block Table

Nephele tracks recoverable memory in *blocks*, fixed-sized regions of memory that are persisted atomically. Most functionality is based around the *block table*, a persistent data structure that keeps track of each allocated block in the system. This table is replicated using the same mechanism as any other recoverable memory. The first page of the block table is always stored with a constant identifier in remote memory. After that, the block table is self-describing and can be recovered using the mechanisms described below.

Each entry in the block table contains the virtual address of the active block on the client and a remote identifier that can be used to identify the block in remote memory.

4.3.2.3 Initialization and Recovery Procedure

When `neph_cfg_create()` is called the first time, it initializes the block table to an empty state and persists it to remote memory. When recovering, `neph_cfg_create()` fetches the first block of the block table. The block table is walked from start to finish, fetching each block as it goes. Even if the block table takes up multiple blocks, each one is fetched in order, ensuring that all data can be found eventually. When Nephele fetches a remote block, it must ensure that it is loaded to the same address it was at before failure, otherwise pointers in the data would no longer be valid. The original address is read from the block table and then allocated using the *mmap()* system call. To ensure that these addresses are always available, Nephele requires that any OS address space layout randomization be disabled, and that `neph_cfg_create()` be called before any other local allocations.

4.3.2.4 Allocation

To ensure that memory is recoverable, the user must allocate it using a special `neph_alloc()` function. The `neph_alloc()` function allocates memory both locally and on the remote node. Recoverable memory is allocated from pools of virtual pages that are then checkpointed on a page granularity rather than tracking individual allocations. Any modifications to the local pages allocated by `neph_alloc()` are automatically detected and copied to the remote node at commit time. Detection is achieved through the use of *mprotect()*, a Linux system call that can be used to make the application take an interrupt when a page is written to. Our interrupt handler then marks the page as changed, removes the memory protection, and returns. This means that Nephele needs to be involved only in the first modification to a page.

4.3.2.5 Marking a Point of Consistency

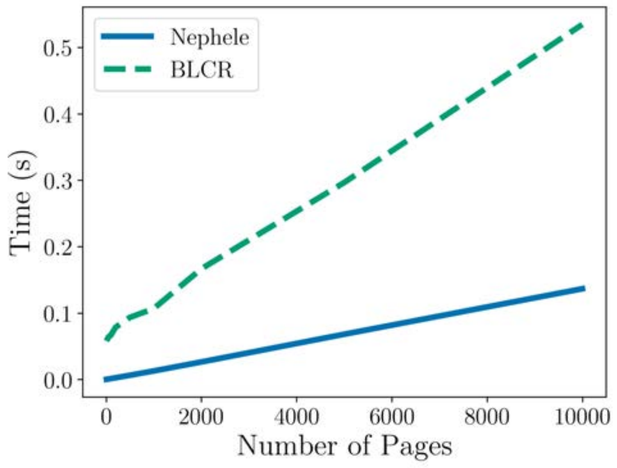
The user is required to identify points in their code where the state of recoverable memory is considered consistent. This means that recovery is possible from that particular state. `neph_txn_commit()` can be called at these points to ensure that memory is atomically persisted. Upon entering `neph_txn_commit()`, Nephele goes through the list of changed pages and copies them to a shadow page in remote memory. This ensures that a consistent version of memory is always available, even if the client crashes during checkpointing. When all the pages have been copied, Nephele commits the changes to remote memory.

4.3.3 Evaluation

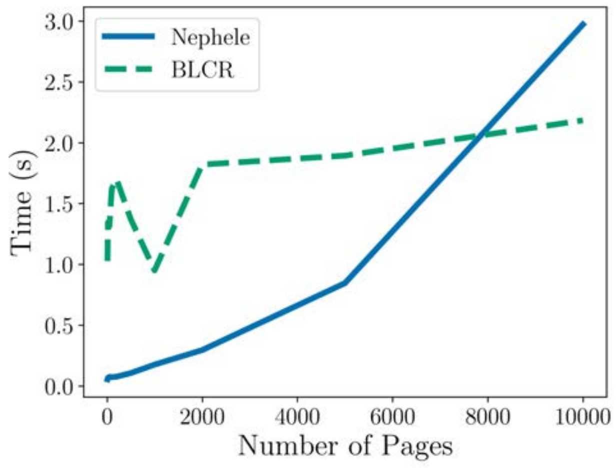
4.3.3.1 Microbenchmarks

I evaluate Nephele first with a simple microbenchmark. For commits, I allocate and modify a variable number of pages. I report the time for each checkpoint to complete. For recovery, I measure the time before the first useful instruction executes after recovery. I compare Nephele against BLCR, a common process checkpointing tool from the high-performance computing community.

Figure 4.20a shows the results of the commit microbenchmark. At the smallest page count, Nephele's commit time is around 150 µs. Time then scales linearly with page count, with a slope of approximately 14 µs per page. Commit time is dominated by the low level remote memory interface since page change tracking only requires maintaining a simple list of changed addresses. BLCR sees significantly longer commit times in all cases with roughly



(a) Commit



(b) Recovery

Figure 4.20: Microbenchmark results

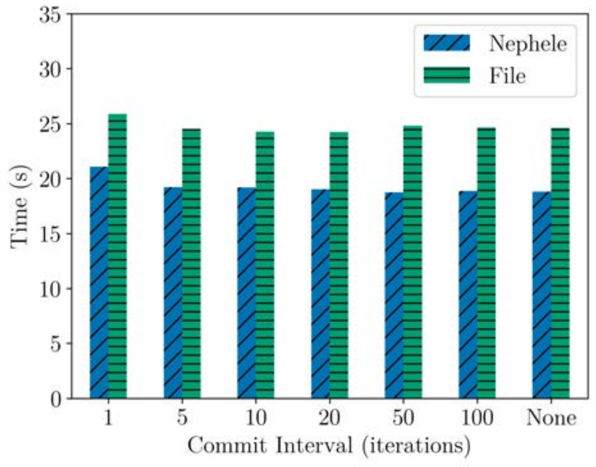
60 ms even when we have not allocated any test pages. Commit times then increase linearly with the number of pages by 47 ms per page.

Figure 4.20b shows the results of the recovery micro-benchmark. Recovery is significantly slower than commit with an initial latency of roughly 50 ms. Recovery time is dominated by Infiniband overheads, particularly page registration with the Infiniband driver. BLCR recovery times are noisy, but take well over one second to complete regardless of process size. Nephele experiences little constant overhead compared to BLCR because it does not need to replicate process state precisely. As I increase the number of pages, BLCR begins to outperform Nephele due to the poor scaling of the remote memory interface.

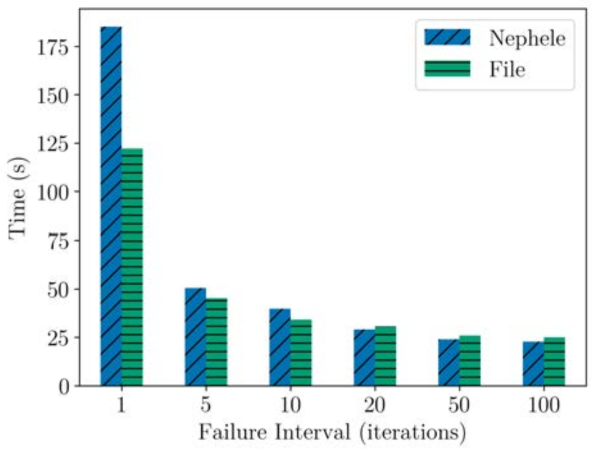
4.3.3.2 DGEMV Benchmark

DGEMV is an iterative dense matrix-vector multiplication benchmark. This benchmark is designed to be ammenable to checkpoint/restart. The matrix is kept constant between iterations so there is very little new data to save on each checkpoint. The only dynamic state, the vector, is also relatively straightforward to serialize. Upon recovery, the DGEMV benchmark loads the output vector and current iteration count from recoverable memory and restarts the computation. I compare Nephele against a manual serialization scheme using a local SSD. In this experiment, I run 100 iterations using a matrix of double-precision floating point numbers with dimension ($100M \times 100$). I then vary the number of iterations between checkpoints and failures.

Figure 4.21 shows the results of this experiment. I first observe that the commit frequency does not have a significant impact on runtime for this benchmark due to the very small dynamic state. Nephele consistently outperforms the manual serialization approach



(a) **Commit** The number of iterations between commits increases from every iteration up to "None" where no commits were taken.



(b) **Recovery** The number of iterations between induced failures increases from every iteration up to every 100 iterations.

Figure 4.21: DGEMV benchmark results

due to the lower latency infiniband interface. When simulating failures, however, we see a much stronger effect due to the overhead of starting a new process. At all but the most extreme failure rates, Nephele has similar performance compared to the manual serialization approach. As I presented in the microbenchmark, most of the startup time for Nephele is due to inefficiencies in the Infiniband interface. Overall, Nephele performs similarly to a tailored manual serialization approach.

4.3.3.3 Genome Assembly Benchmark

I also evaluated the genomics benchmark, Gen, described in §4.2.5. To summarize, this benchmark involves building a large hash table from short DNA sequences and then repeatedly probing to find matching substrings. The benchmark tracks which phase it is in and the current state of the hash table in recoverable memory. During the build phase, recoverable memory also includes the index of the last input that has been loaded. When probing, it includes the current substring, position within the substring, and the list of matches found so far. To recover, the benchmark first checks which phase it is in and calls the appropriate function. Since all working state was stored in recoverable memory, the build and probe functions can begin processing immediately without any additional recovery code.

Unlike the DGEMV benchmark, Gen uses a complex data structure that is not easily serialized. Furthermore, each iteration changes only a small subset of the state, but that subset is spread around the large hash table. In Gen, I use a 100 MB dataset with 8 million probing iterations. Larger problem sizes would increase the initial recoverable memory size,

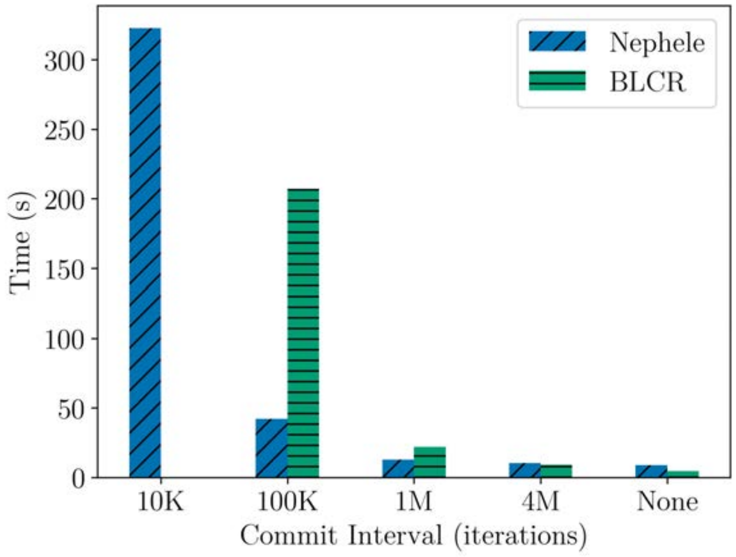


Figure 4.22: Genome commit performance results. They number of iterations between commits increases from every 10K up to "None" where no commits were taken.

but the rate of state change is independent of scale. I would expect Nephele's commit times to remain roughly constant after the first commit as we increase the problem size.

Figure 4.22 shows the result of an experiment where I vary the interval between commits from a very aggressive rate of one checkpoint per 10K iterations down to no checkpointing at all. Since the state is difficult to track and serialize, I compare against the general-purpose checkpoint/restart framework BLCR. For aggressive commit rates, Nephele greatly outperforms BLCR. This is due to Nephele's efficient change tracking mechanisms. While BLCR must checkpoint the entire process on each iteration, Nephele only copies the pages of the hash table the have been updated. At less aggresive commit rates, overheads in infiniband initialization overwhelm any savings from efficient replication.

4.3.4 Nephele Conclusions

Nephele takes advantage of application insights to create a highly efficient checkpoint/restart system. For Gen, Nephele was over 4x faster than a general purpose system. While I observed performance overheads from our network interface, many of those limitations were solved in later versions of the atomic replication library [55]. The true limitation of this approach was not in the underlying technologies, but in the significant effort it required from users. For some simple applications it can be easy to identify critical and ephemeral state. For others, it may require significant changes to the application design. Likewise, recovery code can be complex, particularly for high level languages like Python. These changes may introduce some essential complexity to applications, but most of the new complexity is accidental[2]. In other words, Nephele may be challenging to retrofit into an existing application, but it is much easier to include in new designs.

4.4 Final Thoughts on Physical Disaggregation

In this chapter, I provided a largely physical perspective on disaggregation for memory. With the page fault accelerator, my goal was to increase the amount of memory available to applications with minimal changes to their structure. In Nephele, I used an Infiniband based library to replicate memory changes automatically to memory in a different failure domain. From the perspective of the taxonomy presented in §4.1, I fixed one dimension and explored options along the other. For the PFA, this meant adding hardware acceleration to a largely `implicit` interface. It did not fundamentally change the behavior of existing techniques like Infiniswap, it simply accelerated critical phases of their execution. While this allowed me to make large improvements in performance, the absolute performance remained quite poor in many cases. In Nephele, I picked a middle ground along the `implicit/explicit` axis, but assumed a fixed hardware environment. This more `explicit` approach leveraged application insights to provide significant performance improvements. A major limitation of both these techniques is that they do not address the question of how disaggregated memory should interact with a larger system. How should it be allocated and deallocated? Which remote memory locations should be used? How do applications access and share remote memory?

The answers to these questions will constrain our choices for physical disaggregation. If our system interface demands fault tolerance for user memory, the PFA would need to be augmented with replication and consensus protocols. It is possible that these features would make the hardware implementation infeasible. Since the PFA is based on Unix's concepts of long-running processes, it is not possible to dynamically deallocate resources from a user. Nephele is more flexible. It allows the system to free a process's local resources by terminating and restarting it from a checkpoint. It also does not require any particular hardware, though its performance is sensitive to the underlying network interface.

The reason that Nephele was so flexible was that it presented users with a logical view of the system that is well suited for disaggregation. The PFA struggled because its logical model was based on single-node abstractions that don't map well to remote memory. In the next Chapter, I will continue to explore this insight by considering how a disaggregated system should be presented to users *logically*.

[2]Fred Brooks introduced these terms in his book "No Silver Bullet". They refer to complexity that is fundamental to the problem (essential) or due only to details of the available tools (accidental).

Chapter 5

Logical Disaggregation: Serverless

In Chapter 2, I described a number of available interfaces to warehouse-scale computers. These interfaces present a logical view of the system. That is, they represent what the user *thinks* the system is, independently of how it is physically deployed. In Chapter 4, I described a number of novel ways to physically deploy a system while assuming a fixed logical interface. In this chapter, I explore the interplay between these two concepts with a particular focus on disaggregated compute resources. I first show in §5.1 how a distributed operating system can address underutilization with high performance by using low-noise operating system environments called *unikernels*. I then present a Xen-based operating system that coordinates the scheduling of these unikernels to further reduce noise as we distribute applications across a warehouse-scale computer (WSC). I go on to argue in §5.2 that the logically disaggregated serverless computing model can provide higher utilization while enabling rapid innovation in hardware. I demonstrate this property by designing a new serverless interface to GPUs that reduces cold start times by over 20x and sustains 50x higher throughput than traditional approaches in a multitenant setting.

5.1 Noisy Performance in Distributed Operating Systems: WabashOS

Distributed operating systems seek to provide a familiar single-node abstraction to users of large clusters. Logical disaggregation is provided using a wide range of techniques ranging from microkernel style message passing as in Mach [177], to POSIX-compatible interfaces like LegoOS [246]. In this section, I will describe an operating system called WabashOS that my collaborators John Kubiatowicz, Juan Colmenares, Steven Hoffmeyer, Eric Roman, Matthew Francis-Landau, Sven Schwermer, and I proposed (Figure 5.1). WabashOS is based on the Tessellation distributed operating system [69]. Rather than using a bespoke kernel, WabashOS uses the Xen hypervisor to provide heterogeneous environments called *cells*. Like Tessellation, we envisioned a resource allocation broker and monitor that would tailor resource allocations to cells to meet performance goals with the minimum allocation.

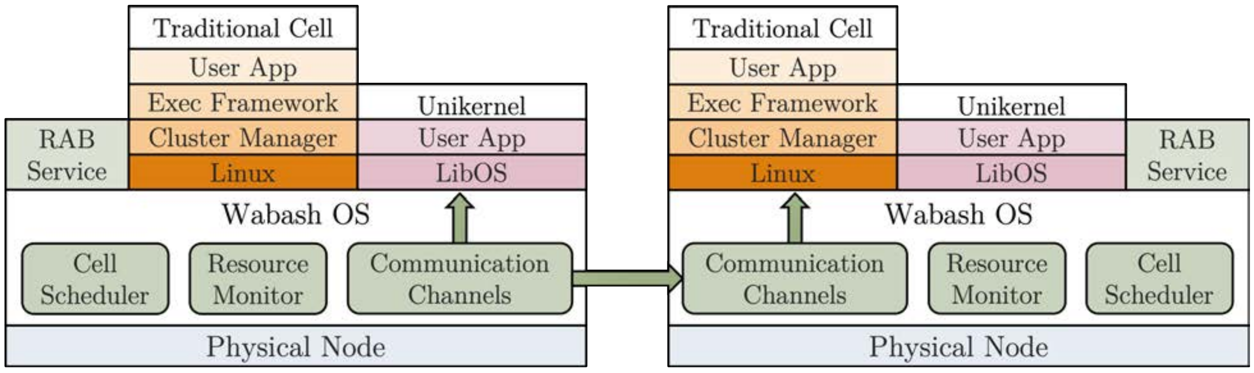


Figure 5.1: **WabashOS Design** WabashOS is designed around a per-node kernel based on Xen, heterogeneous per-process execution environments called *cells*, and a distributed resource allocation broker (RAB) that handles adaptive resource management. Processes can communicate via built in communication channels.

Larger applications would be constructed from groups of these resource-optimized cells. In other words, WabashOS logically disaggregates resources by providing tailored allocations and resource-specialized cells. By focusing on a single resource, these cells were less likely to lead to stranding. Cells communicated through channels facilitated by the kernel that enabled physical disaggregation. This flexibility would allow WabashOS to make placement decisions based on expected communication patterns and resource availability. Barrelfish and Popcorn Linux provide other examples of using resource-specialized kernels for distributed heterogeneous computers [36, 28].

Though not a full end-to-end implementation, WabashOS provided a framework for several research projects. In this section, I will describe two of my efforts to improve utilization of compute resources in WabashOS while enabling high performance. The key insight in these projects was that while distributed operating systems present a familiar single-node logical interface, they can have very different performance characteristics. In the case of WabashOS, I was interested in the impact of poor performance predictability (i.e., *noise*) on coordination-heavy applications. In §5.1.1, I describe the impact that noise can have on application performance. I then describe how I used unikernels to reduce the noise experienced by a single cell. Finally, §5.1.4 describes how multiple cells can be co-scheduled to minimize noise while still enabling multitenancy.

5.1.1 The Impact of Noise

On modern systems, multiple identical runs of an application may take varying amounts of time. This may be caused by variations in hardware performance, because of non-deterministic scheduling decisions in the OS, or from interference from other applications or the OS itself [37]. Collectively, I refer to this non-deterministic performance as *noise*.

The impact of noise on application behavior may not be apparent for long runs because the noise does not consume a large fraction of total application time, and affects all threads equally on average. However, applications that synchronize frequently can be impacted. For example, the Linux kernel has a 1000 Hz timer tick that lasts for approximately 5 µs. This results in a modest 0.5 % overhead. However, suppose that an application is run on an Intel Knights Landing many-core processor with 72 threads, and all threads synchronize on a barrier. Even if all threads make equal progress between barriers, there is a 30 % chance that at least one of them will be interrupted by the timer tick during the barrier[1]. If barriers are frequent, this overhead can be significant. Such frequent barriers are common in scientific applications such as simulators.

Indeed, system noise has been identified as a major source of performance degradation in several HPC deployments where the number of threads can be in the tens or hundreds of thousands [37, 209, 38]. A more detailed analysis of the issue can be found in [89]. In the cloud and web-scale communities, scheduling uncertainty, interrupt routing, and interfering applications have all been cited as contributing to tail latency [154, 152]. Logically disaggregated systems like WabashOS exacerbate this issue by decoupling applications into resource-specialized tasks, leading to potentially many communicating threads.

5.1.2 Common Noise Management Techniques

5.1.2.1 High-Performance Computing

In high-performance computing (HPC), ultimate performance is often favored over utilization. Applications also tend to be more self-contained and specialized than in cloud settings. These properties enable more aggressive and disruptive approaches to noise mitigation.

One example of this trade-off is the emergence of low-noise operating systems [139, 126, 100]. These systems have high performance for core application components, but may not support every aspect of the application. Kitten is a minimal OS designed to run in a co-kernel mode, where both Linux and Kitten run bare-metal on disjoint resources within the same physical node [145]. This configuration allows two cooperating applications, each with different OS requirements, to run on the same machine. A highly tuned simulation code may run on the low-noise Kitten cores, while a visualization application processes its output on top of Linux. Some efforts have shown that Linux can be made to match the performance of specialized operating systems [249].

At the cluster level, HPC deployments typically use a batch scheduling interface such as Slurm [297]. Batch scheduling ensures that applications receive their full resource allocation up-front to avoid unpredictable performance from dynamic resource allocation. Even with static allocations, a job may experience noise from other nearby jobs in the system, necessitating even more careful and exclusive placements [38]. These long-running, all-or-nothing, static allocations can lead to poor utilization with both idle and stranded resources [176].

[1]Assuming that per-core interrupt timers are independent, and the probability of any one core being in an interrupt is 0.5%, then $P(interrupted) = 1 - 0.05^{72} = 0.30$

5.1.2.2 Cloud Computing

Cloud applications are often designed to support online user interactions. As such, they are primarily concerned with worst-case performance, called *tail latency*. These are measured in terms of percentiles and may be associated with an explicit tolerance called a service-level objective (SLO). Violating an SLO is considered unacceptable while latencies below the SLO provide little benefit. Li, et al., survey the sources of noise in large scale interactive applications [154]. These sources include queing delays and head of line blocking, background tasks, and hardware behaviors like NUMA or dynamic power management.

There are many strategies for mitigating tail latency in the cloud. In “The Tail at Scale”, Dean and Barroso describe the impacts of noise on Google’s applications and propose a number of mitigations [75]. Some techniques mirror those seen in HPC environments. These include SLO-aware scheduling, physical isolation for sensitive jobs, and careful scheduling of background tasks. However, they also acknowledge the inevitability of noise and design applications to tolerate noise when it does occur. These applications may hedge requests by sending the same request to multiple servers and using only the fastest response. They also factor applications into small partitions that can be easily moved between servers to balance load. Tail latency is also critical when serving machine learning models, leading to specialized SLO-aware model-serving systems [105, 230].

5.1.3 Unikernels

In Kitten, OS heterogeneity was provided by the *split kernel* approach where multiple operating systems ran on physically disjoint and persistent partitions of a physical machine. For highly predictable and long-running workloads, this approach can be reasonable. The difficulty arises when we introduce multitenancy and heterogeneity. An ideal partition between Kitten and Linux for one application is unlikely to be ideal for another. Even within an application, different phases of execution may have differing resource and operating system requirements.

In datacenter applications, hypervisors have been used extensively to provide more flexibility in OS deployment. As this approach became more ubiquitous, operating systems began to employ virtualization-native optimizations such as virtio in Linux [125] or Xen’s paravirtualized interfaces [29]. These interfaces allow the guest OS to cooperate with the hypervisor on virtual memory management, trap handlers, and IO device management rather than emulating low-level hardware interfaces. Even more radically, some began designing operating systems that *only* ran on hypervisors. These operating systems are called *unikernels* [164, 132, 84, 95]. In WabashOS, we took advantage of these trends by reconceiving of hypervisors as an OS kernel, while unikernels provided the process abstraction. One advantage of this approach is that users are free to choose any guest operating system they wish, from general-purpose Linux to a highly specialized unikernel.

5.1.3.1 Specialization for Unikernels

I focused on one particular unikernel called *Rumprun* [132]. Rumprun is a framework for generating unikernel instances that are tailored for a particular application. Unlike a general-purpose OS, Rumprun is intended to run only a single application with a known set of dependencies and required features. This approach provides a lightweight and low-noise environment for applications. To reduce noise even further, I worked with Juan Colmenares and Steven Hoffmeyer to build a custom low-noise scheduler for Xen. Rather than considering all CPUs simultaneously, our scheduler partitioned the available resources into dedicated pools with differing policies. In this section, I evaluate our lowest noise policy, called *Batch*, that simply dedicates physical cores to virtual cores statically and never migrates or preempts the virtual machine. This drastically reduced the noise contributed by Xen in the guest operating systems. In §5.1.4, I describe an alternative policy that enables multitenancy while maintaining a low-noise environment.

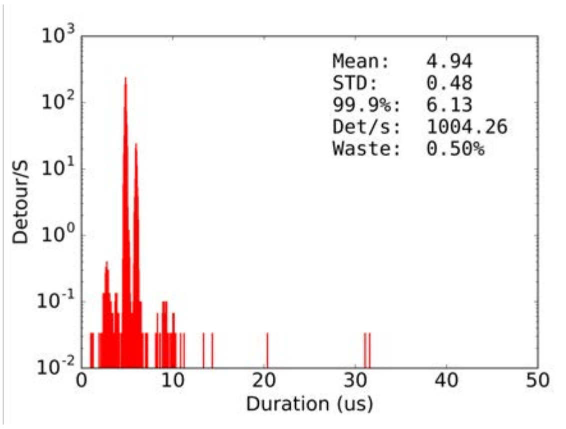
5.1.3.2 Evaulation of Unikernel Noise

I measured noise using the Selfish Detour benchmark from the Netgauge benchmarking suite [112]. Selfish Detour repeatedly polls the CPU's real-time clock and records any deviations from the expected poll duration (called *detours*). I report the full histogram of these detours as well as the time contribution of detours on end-to-end runtime of the benchmark, here called *waste*. In all cases, I ran the benchmark on a dedicated CPU core with all other configurable OS tasks pinned to a separate CPU socket. I disabled symmetric multi-threading and dynamic frequency and voltage scaling on the test machine. Table 5.2 shows the details of my experimental environment. Figure 5.3 shows the result of this experiment.

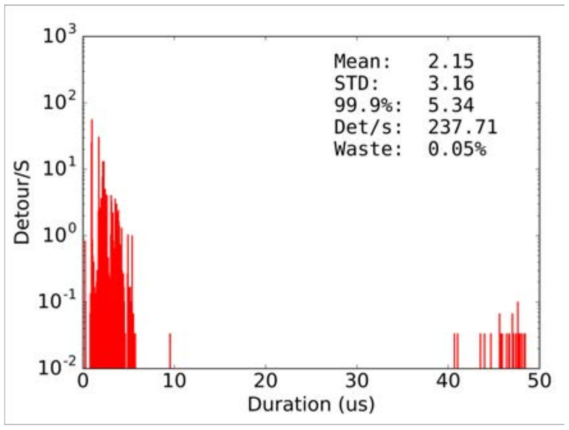
CPU	2x Intel Xeon E5-2697 v3
CPU Frequency	2.6 GHz (dynamic frequency disabled)
#Cores	14 per socket
#Threads	1 per core (SMT disabled)
Memory	64 GB DDR4-2133
Xen Version	4.4.2
Linux Version	3.16

Figure 5.2: Evaluation platform details

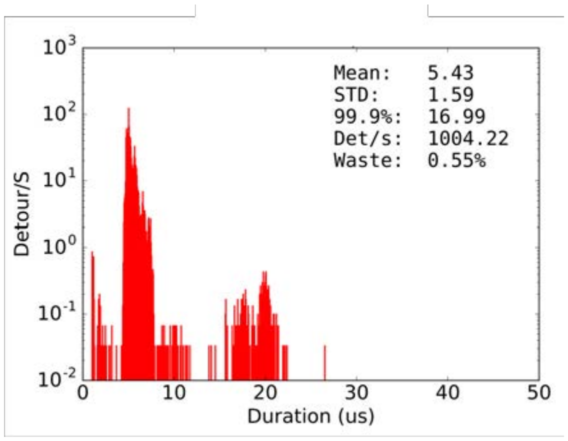
In Figures 5.3a and 5.3b, I ran the benchmark with no other user processes on the system. This presents an optimal scenario for low-noise execution. However, we see significant noise on Linux with 1004 detours measured per second and an average duration of about 5 µs. The peak at 5 µs corresponds with the Linux scheduling quantum while other detours come from Linux background events such as RCU management (see [173]) and variability in the runtime of the scheduler itself. Rumprun has significantly lower noise with only 238 detours



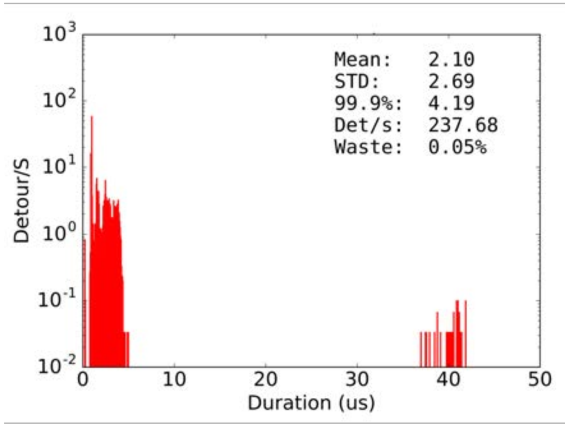
(a) Linux Alone



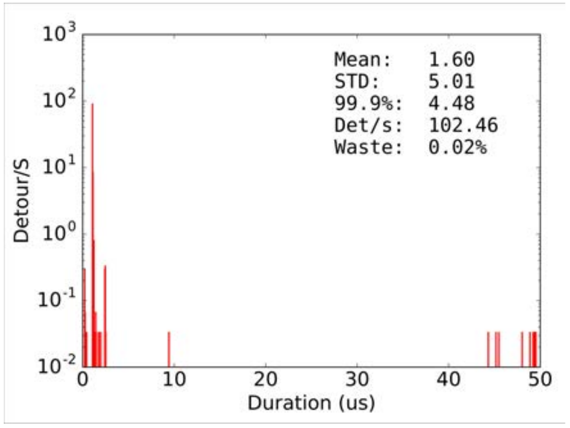
(b) Rumprun Alone



(c) Linux with a competing workload



(d) Rumprun with a competing workload



(e) Rumprun with a competing workload using my custom Xen scheduler

Figure 5.3: Results of the selfish detour benchmark on different operating system environments.

per second and a lower average duration of 2.15 µs. The cluster of detours between 40 µs to 50 µs corresponds with very rare Xen-specific events.

I then configured the experiment to include a competing workload on a separate CPU socket (Figures 5.3c and 5.3d). The competing workload consisted of a 10-way parallel build of the Linux kernel. For the Rumprun experiments, this workload was run in a Linux virtual machine on top of Xen. For Linux, the workload was run as a set of processes pinned to the appropriate cores. This workload interacts with the OS frequently and uses a significant amount of memory and CPU cycles. The results for Linux show a similar frequency of detours to the unloaded case, but the duration of those detours increases signicantly from 6.13 µs at the 99.9 % tail to 16.99 µs. This is concerning because the selfish-detour benchmark runs on dedicated cores, requires no OS services, and is completely CPU-bound. The interference is instead due to synchronization overheads in the Linux kernel that slow regular events like timer interrupts, even on unrelated cores. Rumprun on Xen sees no significant change in behavior due to the competing workload.

Finally, Figure 5.3e shows the behavior of our custom scheduler on Rumprun with a competing workload. Since our scheduler gave complete control of the CPU to the guests, the only major source of detours comes from a 100 Hz internal rumprun timer used to perform background operations. There are also a number of very rare detours from Xen and hardware variability. In all, we see only 102 detours per second with low mean and tail durations of 1.6 µs and 4.48 µs, respectively.

5.1.3.3 Unikernel Conclusions

These results demonstrate the challenges in using a general purpose operating system for high-performance applications. In this case, I had little need for most operating system services in my benchmark application and would not benefit from the sophisticated shared services like page caches or multitasking schedulers provided by Linux. Applications on a multi-tenant WSC will similarly have little need for such shared services at the node level. Unlike other low-noise operating systems, our approach also enabled heterogeneity of operating system services. The selfish-detour benchmark had few dependencies and a low-noise requirement while the competing workload required a full-featured operating system but would not have benefited from low noise. This heterogeneity also presents a more flexible interface for logical disaggregation where execution environments can be specialized for individual resources.

5.1.4 Gang Scheduling

In the previous section, I showed how Xen could be modified to support low noise unikernel environments. Low noise is an important goal, but that performance predictability came at the cost of resource utilization. Applications ran to completion on dedicated resources, even if they became blocked on IO or synchronization. Michelogiannakis, et al., report median CPU utilization of only 50% for traditional CPUs and 75% for the specialized Knights

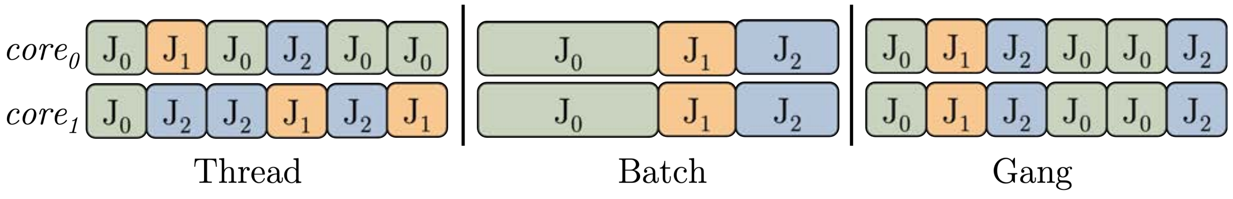

$core_0$
$core_1$
J_0 J_1 J_0 J_2 J_0 J_0
J_0 J_2 J_2 J_1 J_2 J_1
J_0 J_1 J_2
J_0 J_1 J_2
J_0 J_1 J_2 J_0 J_0 J_2
J_0 J_1 J_2 J_0 J_0 J_2
Thread
Batch
Gang

Figure 5.4: Example execution for different scheduling policies. Thread scheduling considers each job's threads independently. Batch runs an entire job to completion while Gang runs all threads concurrently, but interleaves execution between different jobs.

Landing processors on the Cori supercomputer [176]. Specialized compute units like the Knights Landing see particularly variable utilization. At the 40th percentile, we see less than 10% utilization. Another consequence of this static allocation is that users in a multi-tenant environment may need to wait for previous jobs to finish before theirs can start. This is frustrating for users, particularly those with short-running jobs. It also means that application pipelines with varying resource needs at different stages may become blocked, even if the system has enough resources for one phase but not others. In this section, I evaluate the impact of scheduler flexibility on noise and utilization. I compare traditional algorithms against a middle ground, called Gang Scheduling, that we implemented in WabashOS.

5.1.4.1 WSC Scheduling Background

There is a broad range of scheduling strategies in common use that trade off between utilization and application performance. Figure 5.4 shows three categories of scheduling algorithms that I considered in my work. Per-thread scheduling is common in cloud and general-purpose computing. Examples include Linux's Completely Fair Scheduler, Xen's default Credit algorithm, or even real time strategies like Earliest Deadline First [13, 294, 259]. This strategy maximizes utilization by allowing threads to run on any idle resources, regardless of other threads in the application. While this uses resources effectively, communicating threads are not guaranteed to be scheduled simultaneously. This can lead to poor performance for jobs with frequent synchronization between threads. The HPC community is more concerned with ultimate performance of a single job than pure utilization. The most common HPC strategy is a batch scheduler that runs every job from start to finish on dedicated resources [297, 270]. This strategy ensures that all threads can synchronize without waiting for the scheduler. Gang Scheduling aims for a middle ground. It ensures that every job thread is scheduled at the same time, but it interleaves jobs in time to ensure that all jobs make progress [200, 124]. It enables high performance for synchronization-heavy jobs while still allowing multitasking. While gang scheduling is often associated with HPC environments, it has also been used in enterprise settings. For example, VMWare's vSphere product approximately co-schedules VMs to minimize clock drift [265].

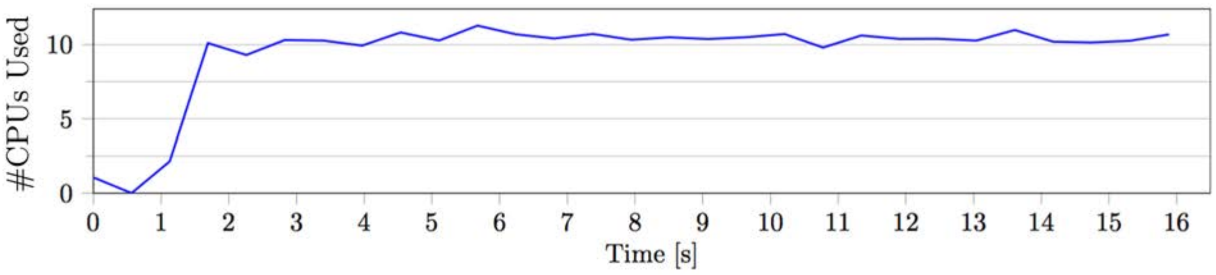


Figure 5.5: CPU utilization of the CoEVP benchmark over time on a 14 core CPU.

5.1.4.2 Gang Scheduling in WabashOS

I worked with Steven Hoffmeyer and Juan Colmenares to add support for gang scheduling to WabashOS. Matthew Francis-Landau, Sven Schwermer, and I then evaluated this design with a microbenchmark and two scientific applications: CoEVP and CoHMM.

5.1.4.3 Benchmarks

Microbenchmark Gang scheduling ensures that threads can communicate without waiting for each other to be scheduled. It also ensures that all threads make similar progress over time, a valuable property for applications with frequent barriers. In this section, I use a microbenchmark to measure how much skew different scheduling policies introduce between threads. The benchmark repeatedly runs a simple CPU-bound loop for 100 ms on 14 threads. Ideally, every thread would make equal progress to minimize the time spent waiting for barriers. For this benchmark, I require that all threads remain within one iteration of each other. If any thread lags by more than one iteration, I consider it to have missed its deadline. I run the benchmark for 28000 iterations with a varying number of concurrent applications and measure the number of missed deadlines.

CoEVP CoEVP simulates the deformation of a tantalum cylinder fired at a solid wall [80]. It simulates this scenario at multiple levels of detail by combining many independent fine-grain simulations into a large-scale result. These fine-grain simulations use OpenMP to parallelize the algorithm. The results of these fine-grain simulations are collected into an embedded database that is later queried by the coarse-grain simulation before beginning a new phase. It does not perform significant IO and is primarily compute-bound. Figure 5.5 shows the CPU utilization of CoEVP over time when run with 14 cores. CoEVP has consistent CPU utilization, but does not scale beyond 10 cores.

CoHMM CoHMM simulates shock propagation in a copper plate after it is hit by a projectile [67]. Like CoEVP, CoHMM performs a multi-scale simulation with the bulk of computation spent on fine-grained simulations. CoHMM adapts the granularity of simulation based

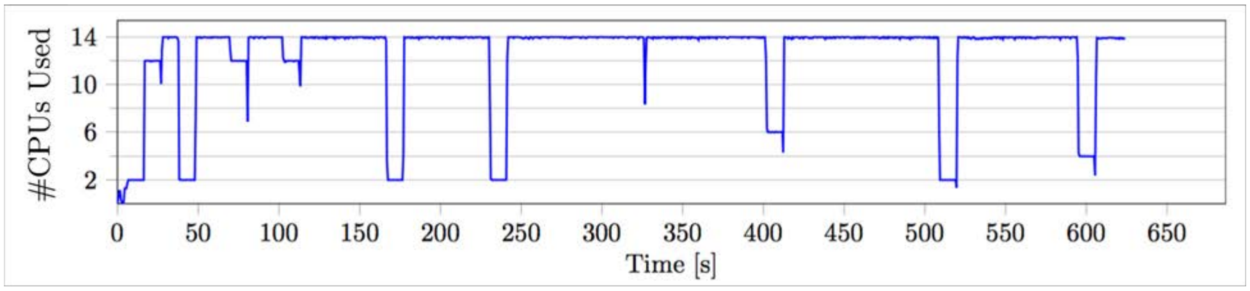


Figure 5.6: CPU utilization of the CoHMM benchmark over time on a 14 core CPU.

on previous results and may re-use the results of previous simulations. Intermediate results are stored in a dedicated key-value store that is queried periodically. In our experiment, we used Redis in a Linux cell running on a separate CPU socket from the main CoHMM application. Figure 5.6 shows the CPU utilization of CoHMM over time on a 14-core cell. Note that it is often able to utilize all 14 cores, but experiences periods of idleness at simulation epochs where it must perform IO to the key value store.

5.1.4.4 Gang Scheduling Results

I evaluated the Gang scheduler against Xen's default per-thread scheduler called *Credit* [294] and our Batch policy that runs jobs serially on dedicated resources. The experimental setup is the same as described in the Unikernel experiments, but I used only Linux-based cells.

Microbenchmark Figure 5.7 shows the results of the microbenchmark. Regardless of the number of concurrent cells, Gang scheduling ensures that all threads make equal progress. An application with frequent barriers could expect to make predictable progress between threads. Xen's Credit scheduler interleaves threads from multiple cells opportunistically, leading to significant differences in per-thread runtime. The Credit scheduler missed 86 % of all deadlines, even at a conservative 100 ms period.

Macrobenchmarks I now evaluate the macrobenchmarks by running multiple concurrent cells and recording the total runtime for all cells to complete. For reference, CoEVP takes approximately 13 s when run in isolation, CoHMM runs for 10 min.

I begin my analysis with CoEVP in Figure 5.8a. At low levels of concurrency, Gang scheduling behaves similarly to the per-thread scheduler. As concurrency increases, however, the advantages of a per-thread scheduler become apparent. Recall that CoEVP cannot effectively utilize all cores on our system, leaving 2 cores idle. Since we have a fixed-size physically aggregated CPU, this leads to stranded resources with the Gang and Batch scheduling approaches. The per-thread scheduler can effectively utilize these cores to run threads from other cells. Surprisingly, Gang scheduling slightly outperforms Batch at all levels of concurrency. I hypothesize that this advantage comes primarily from an overlap of WabashOS

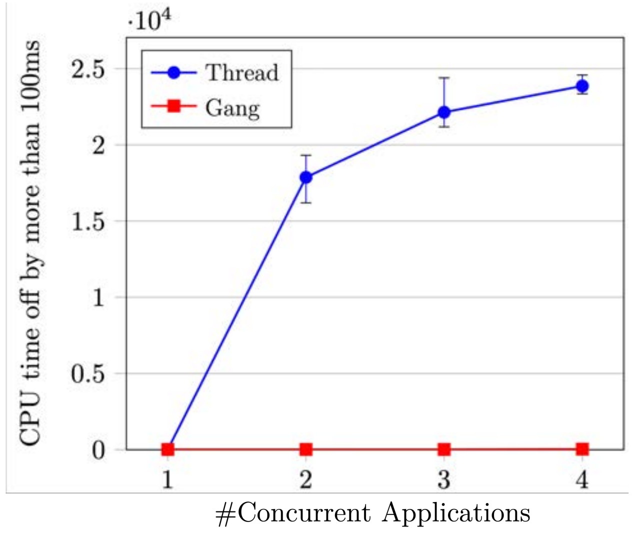


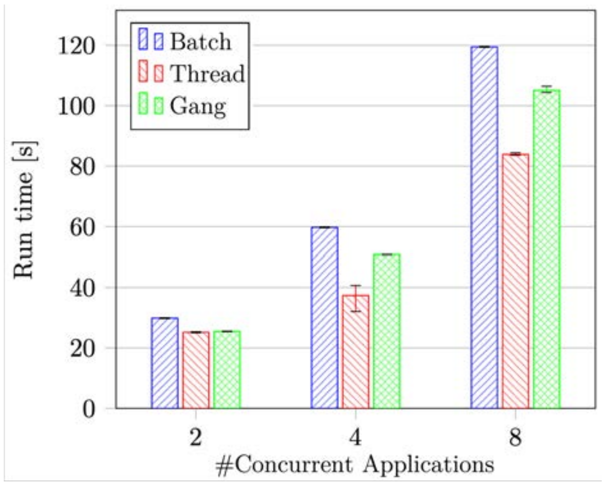
Figure 5.7: Results from the synchronization microbenchmark. I report the number of missed deadlines under Gang and per-thread scheduling as we increase the number of competing workloads. The benchmark was run for 28000 iterations.

tasks such as process loading or terminal outputs. Although the application code sees similar behavior in both cases, the Batch scheduler cannot even begin loading the next application's cell until the previous cell terminates.

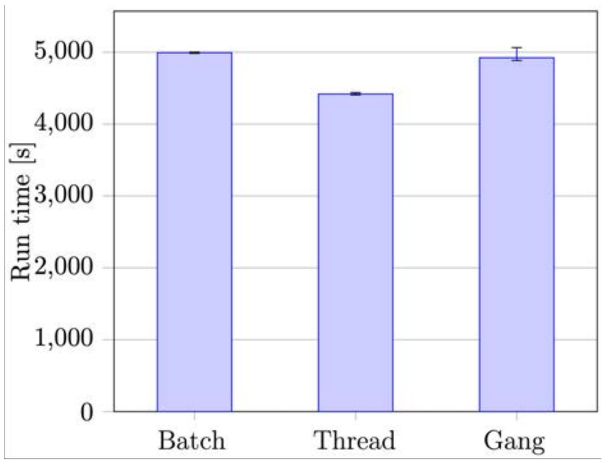
CoHMM is a longer running application with more complex system interactions. Figure 5.8b shows the behavior of CoHMM under different schedulers with 8-way concurrency. While the per-thread scheduler still outperforms the other strategies, it does so by a narrower margin than with CoEVP. Where CoEVP had poor internal CPU utilization, CoHMM is better able to utilize all cores. Since CPU utilization is quite good under all scheduling policies, there is little room for the per-thread scheduler to improve. Gang and Batch also perform very similarly on this long-running benchmark, supporting my hypothesis that cell start up overlap contributed to Gang's advantage in CoEVP.

5.1.5 WabashOS Conclusions

Distributed operating systems like WabashOS allow applications to transparently span multiple resources across a cluster. With WabashOS, we provided some amount of logical disaggregation through the Adaptive Resource Centric Computing paradigm introduced by Tessellation [69]. Applications were decomposed into resource specialized cells that could be created and destroyed as needed rather than allocating all resources up-front. In this section, I mostly focused on the impact of this flexibility on application performance. While general purpose operating systems and schedulers provide good utilization, they also make



(a) CoEVP runtime as the number of concurrent applications increases.



(b) CoHMM runtime under different scheduling policies with 8 concurrent applications.

Figure 5.8: Macrobenchmark Results

performance less predictable. For parallel applications, this noise can result in poor performance.

Reducing noise, however, also comes with a cost. Gang scheduling ensured that applications saw system behaviors that mimicked the batch scheduled environments they were designed for, but led to stranded resources. This result parallels those seen in Chapter 4 where transparent and familiar interfaces limited our ability to disaggregate and improve utilization. In the case of Unikernels, I made more radical changes to the programming environment, but also saw large reductions in noise without hurting performance. The problem there was that the specialization that reduced noise also reduced generality. Applications wishing to use a Unikernel will likely need to be adapted to that environment and may not find every feature they need.

While not a focus of my work, WabashOS had other limitations from a disaggregation perspective. The cell abstraction allowed for some degree of resource specialization, but it still aggregated multiple resources into a single allocation. Cells got all of their CPU and memory resources up-front and maintained those allocations for their lifetime. State was managed by the application rather than the system, limiting our ability to physically disaggregate memory resources. These unbounded allocations of multiple resources would still lead to the idle and stranded resources we see in serverful environments.

This tradeoff is fundamental. Reaching the full power of physical disaggregation will require re-thinking the system abstractions that we present to users. Retrofitting logical disaggregation onto fundamentally aggregated abstractions will always limit our ability to physically disaggregate. In the next section, I present my work on creating a more fundamentally disaggregated system interface by generalizing the serverless computing paradigm.

5.2 Logically Disaggregated Accelerators: Kernel-as-a-Service

I have already argued in Chapter 2 that serverless computing, and function-as-a-service in particular, provides a strong basis for logical disaggregation. Later, in Chapter 6, I will present how I think serverless and physical disaggregation can be combined into a truly disaggregated datacenter. In this section, I describe work I've already done toward this goal by extending the serverless model to specialized compute resources like GPUs.

Serverless computing has already seen significant adoption. Users get true pay-per-use and providers can quickly reallocate resources to other jobs. However, today's Function-as-a-Service (FaaS) systems remain narrow in scope by focusing on CPU-based workloads running general purpose code on familiar OS environments. To date, none of the major cloud providers offer a GPU-enabled FaaS service. This is somewhat surprising given the growing popularity of application accelerators. Why is it challenging to deploy accelerators like GPUs in a FaaS system? The problem lies in the techniques that make FaaS practical to implement. FaaS functions are small and limited in scope, they consume few resources when not executing, and use easily shared and subdivided resources. Providers are free to kill FaaS containers as needed to free resources, or aggressively cache them to reduce cold starts. Furthermore, providers can mitigate the performance costs of explicit state by maintaining shared caches of their data layer. GPUs upend these assumptions. Unlike CPUs, GPUs are expensive, difficult to share on a fine granularity, and have their own memory that must be managed by the user.

This is not to say that GPUs are fundamentally incompatible with the FaaS model; they simply require different techniques. Unlike general-purpose code, GPU functions (called *kernels*) have limited capabilities and few dependencies which greatly simplifies function startup. GPU functions also have more predictable inputs and outputs, enabling greater optimizations from the FaaS scheduler and data layer.

In this section, I present a truly serverless interface to GPUs, called Kernel-as-a-Service (KaaS), that is able to take advantage of these properties to enable high-utilization of GPUs in the cloud with minimal performance overheads. In KaaS, GPUs become first-class citizens that are directly invoked through a GPU-specific function type (see Figure 5.10a). Rather than explicitly mixing host and device code, users register CUDA kernels that run independently with no user-provided host code. Since the KaaS system is in full control of the GPUs, it is able to explicitly manage device memory and multiplex kernels from many users at a fine granularity.

For compute-heavy tasks, KaaS experiences virtually no decrease in aggregate performance, even when there are far more users than available GPUs. For memory-heavy workloads, KaaS performance degrades gracefully when GPU memory requirements exceed capacity. In my evaluation, I compare KaaS to a traditional FaaS approach that allocates GPUs exclusively to functions during execution. My results demonstrate 50x higher throughput and 16x lower latency for a compute-intensive linear system solver when the number of clients

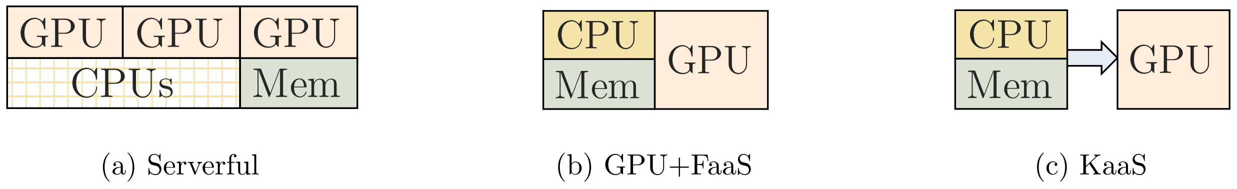


Figure 5.9: Three possible GPU deployment strategies. In a serverful deployment, users get allocated a large collection of CPUs and GPUs for an unbounded period of time. In GPU+FaaS, users run short tasks written in a high level language. These tasks have access to both a CPU and a GPU, but cannot not maintain state between invocations. In KaaS, users submit CPU and GPU tasks separately that each run in a resource-specific environment.

exceeds available GPUs by 4x. Even when total available GPU memory is exceeded by 1.5x, KaaS supports 53x greater throughput and 12.6x lower latency for a memory-intensive deep-learning inference benchmark.

I present the following key contributions:

- A new GPU-native serverless function type, kernel task (kTask), that treats GPUs as first-class citizens.
- Three programming interfaces to KaaS functions: a low-level application programming interface (API), a TVM-based compiler, and a suite of built-in libraries.
- Two scheduling algorithms that take advantage of the unique properties of FaaS and KaaS functions.
- A Ray-based prototype of KaaS with drastically improved utilization of GPU resources in a multitenant environment.

In §5.2.1, I discuss how GPUs are deployed today. §5.2.2 presents the KaaS programming model while §5.2.3 describes my Ray-based prototype. In §5.2.4, I evaluate this prototype against a traditional mixed host/GPU approach with a diverse set of applications ranging from deep neural-network model serving to general purpose linear algebra. I conclude in §5.2.5 with a discussion of some other ways the community has approached the problem of GPU utilization in the cloud.

5.2.1 Cloud GPUs Today

Today, GPUs are exposed to users through a number of allocation schemes that largely mirror the WSC interfaces I presented in Chapter 2. I present the most relevant approaches here (see Figure 5.9).

5.2.1.1 Serverful

Traditionally, GPUs have been deployed in a serverful manner as PCI-E attached cards in a virtual machine (Figure 5.9a). Users are fully responsible for driving enough utilization to justify the added cost. This can be challenging. In an interview with the manager of a genomics assembly and analysis service, I was told that a promising new algorithm had been rejected due to its GPU requirements. While the GPU would have made the system sufficiently fast to offset the added costs when running, the cost of idle resources was deemed too high. In another example, a team had added GPU support to Spark. Results were excellent in some workloads, completing jobs faster and for less money. However, the static resource allocations in most Spark clusters meant that even small delays in a job could result in higher costs than a CPU-only implementation.

5.2.1.2 Service API

Rather than exposing accelerators directly to users, some cloud systems expose *services* that then utilize the accelerators. Of particular note is Microsoft's Catapult system that deploys FPGAs as network endpoints that can be programmed into "FPGA Microservices" [215]. Unlike full-blown servers, FPGAs can be optimized for networked applications with microsecond-level latencies and line-rate processing. For example, the Microsoft Brainwave project places deep neural network models directly on a network of FPGAs that can be addressed from conventional software [65]. Unfortunately, Catapult is neither general purpose nor multi-tenant, limiting its deployment to internal tools or high-level customer-facing services [78]. Broadly speaking, these services lack generality and continue to suffer many of the challenges associated with serverful deployments.

5.2.1.3 Remote GPUs

A lower-level solution to GPU underutilization is to provide network-attached GPUs through *API forwarding* [82, 128, 275]. In this approach, user applications interact with their allocated GPU exactly as if it were local using the same driver APIs. Under the covers, the GPU driver is modified to proxy all requests over the network to a server running on remote machines with idle GPUs. Much as we saw with distributed operating systems, this approach addresses stranded resources, but it does not resolve the other challenges presented by a serverful approach. It says little about device state management or utilization within an application (idle resources). Furthermore, applications written for local GPUs may not account for the increased latency or bandwidth limitations of remote GPUs.

5.2.1.4 FaaS+GPU

Some systems have offered a more serverless FaaS+GPU approach. In these systems, users upload a CPU-oriented function and the provider ensures that the function runs in a container that includes a GPU (Figure 5.9b) [181, 192, 196, 140]. While these approaches present a familiar interface to accelerators, they give away many of the features that make serverless appealing.

As an example, the Ray distributed computing framework supports GPU-enabled remote functions called *tasks*, but requires that applications carefully design their tasks to manually share resources and clean up properly when they finish. In practice, this is difficult to achieve. Indeed, users are advised in the documentation to force Ray to restart workers on each task invocation to ensure resources are properly freed [221]. Rather than using serverless functions, users are often advised to fall back to non-serverless stateful actors to manually manage GPU resources.

Another challenge that arises with this approach comes from cold start mitigation strategies. CPU functions incur a significant startup latency due to container and language runtime initialization. A Python script that simply imports tensorflow and immediately exits takes 1.9 s when the OS buffer cache is warm, a true cold start that must read from disk takes even longer at 6.8 s[2]. To minimize the impact on users, cloud providers often keep function executors allocated in anticipation of a new request [245]. This policy is reasonable because SMT threads are cheap and processes are easily idled. The same policy applied to a GPU would cost at least 60x more since the provider would need to keep the much more expensive GPU idle[3]. While the increased cost of a GPU is justified when fully utilized, both resources provide the same utility when idle: zero.

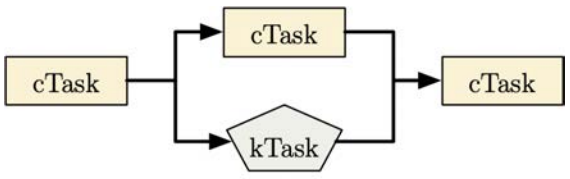
5.2.2 A New Approach: Kernel-as-a-Service

Fundamentally, existing approaches struggle because they cede control of GPUs to applications by tightly coupling host and GPU code. I take a different approach, called Kernel-as-a-Service (KaaS), that explicitly decouples host functions from GPU kernels at the system level (Figure 5.9c). Following the principles of serverless computing, my system takes user descriptions of GPU *functionality* rather than providing GPU *allocations*. This puts the responsibility for GPU memory management and scheduling on the system rather than relying on users to make optimal use of these expensive resources.

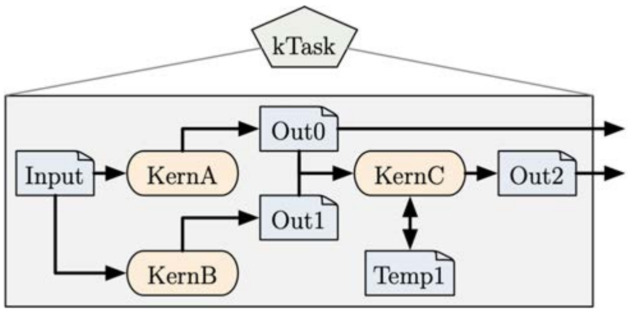
This strategy elevates GPU functionality to a first-class entity, along with traditional FaaS functions and other serverless services (see Figure 5.10a). I will refer to CPU-specialized functions as CPU tasks (cTasks) and GPU-specialized functions as kernel tasks (kTasks). Rather than providing a Python source file or Linux container, kTasks take the form of a graph of CUDA kernels to be executed. Graph inputs and outputs are provided as objects in the system data layer (e.g., keys in a key-value store). Other buffers like intermediate outputs and temporary buffers are described simply by their size. The system is then responsible for ensuring that all required buffers are available in GPU memory before beginning execution of the kernel graph. kTasks are not permitted to dynamically allocate memory or access the data layer, leading to highly predictable resource requirements. Furthermore, kTasks do

[2]Run on an Amazon EC2 p3.2xlarge instance with a general-purpose SSD (gp2).

[3]Based on the capital costs of the 56 SMT core Xeon Platinum 8380 ($10k MSRP[1]) and an Nvidia A100 ($12k[2]).



(a) Serverless graph including both GPU and CPU-specialized tasks. While different task types have different implementations, at the application graph level they all behave similarly; they take inputs, perform computation, and produce outputs.



(b) kTasks are defined as a dataflow graph of individual CUDA kernels with predefined inputs, outputs, and temporary buffers. Inputs and outputs go through the data layer of the broader serverless infrastructure.

Figure 5.10: KaaS introduces a new GPU-specialized function type called a kTask.

not include any host code whatsoever, which simplifies the software environment on KaaS executors and prevents external side effects.

5.2.2.1 Why KaaS?

The KaaS approach uses GPUs exclusively in its programming, and presumably billing, model. This means that users are charged only for the time their code actually ran on the GPU rather than paying for long lived allocations. Users also do not need to allocate an entire server or VM just to use its GPUs. In serverful and FaaS approaches, the user must pay for idle CPU, memory, and disk resources while the GPU is running. Furthermore, KaaS users do not manage the GPU themselves and do not need to (and in fact cannot) maintain any state that might interfere with sharing. Instead, the system is free to manage device memory and allocation at a fine granularity. Like serverless more generally, these properties free users from complex deployment decisions and allow for transparent autoscaling.

5.2.2.2 KaaS Interface

kTasks appear to the rest of the system like any other function. They provide named functionality without any explicit resource allocation, and interact with a common data layer for inputs and outputs. While the KaaS approach is not tied to any specific serverless framework, it does assume a graph of functions communicating through a common data layer. Figure 5.10a depicts a typical application in such a system. Each node in this graph can be implemented in many different ways, and node implementations can be modified or replaced without affecting the rest of the application. While this project focuses on GPU-typed functions, the KaaS approach can be applied to other function types such as deep learning-specific accelerators and specialized systems on chip (SoCs). I will talk about this more in Chapter 6.

While the external interface is assumed to be common to all function types, their internal implementation can be specialized. Figure 5.10b depicts the user's view of a kTask. Users register a library of precompiled CUDA kernels that can then be combined into an execution graph when invoked. These kernels can come from common highly optimized libraries like NVidia's Cutlass linear algebra library [73], code compiled by frameworks like TVM [62], or custom application-specific kernels. Each kernel is provided a set of GPU memory addresses representing inputs, temporaries, and outputs. These data objects are identified using the global data layer's semantics. Users may optionally specify a fixed number of iterations for a particular request. Future work will allow for more dynamic control flow similar to TVM's Relay IR or Dandelion's EDGE graph representation [229, 231].

Nearest-Neighbors Example I now describe an example kTask: N nearest neighbor. In this algorithm, I iteratively multiply an adjacency matrix A by a vector X representing starting vertices. On each iteration, the set of visited vertices in this iteration is accumulated into a vector V representing the set of nearest neighbors. Algorithm 1 describes this more formally.

Algorithm 1 An iterative N nearest-neighbor search over an adjacency matrix A. X_1 contains an initial set of vertices while V_{N+1} contains the final set of nearest neighbors.

```
1: for $i \leftarrow 1, N$ do
2:     $X_{i+1} \leftarrow A \cdot (X_i - V_i)$
3:     $V_{i+1} \leftarrow V_i + X_i$
4: end for
```

Figure 5.11 shows how this algorithm could be implemented in KaaS. The A matrix along with an initial set of starting vertices X_{inp} are passed in as objects in the data layer. Since A is large and constant, it can be cached by the KaaS executor in GPU memory while the smaller X_{inp} may change on each invocation of the algorithm. Some buffers like X_{tmp} and X_{iter} represent intermediate values and can simply be allocated by the KaaS executor. Finally, V_{iter} represents an output of the system and will be written back to the key specified in the KaaS request. The kernels *vsub*, *vadd*, and *matmul* can be implemented in a number of ways, including user-provided code or a built in and optimized library like graphBLAS [45]. Users would likely call this function as part of a larger application with potentially long gaps between invocations. Unlike traditional approaches, users would not need to explicitly provision GPUs in their system and would only pay for the time the GPUs were actually running. In between user requests, the KaaS system would be free to execute functions from other users.

Challenges These benefits do not come for free. The CUDA ecosystem has evolved around a model of locally-attached GPUs. Even multi-GPU workloads require host involvement to

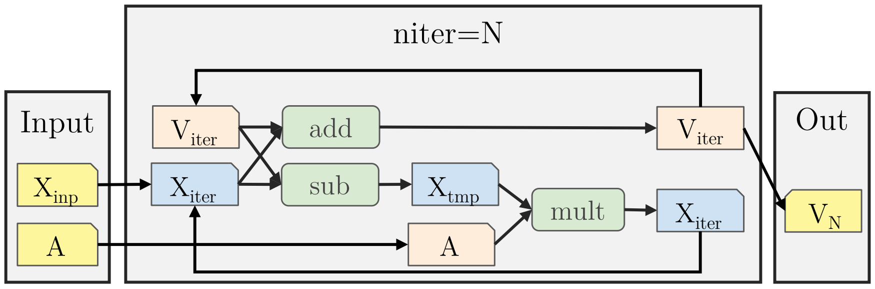


Figure 5.11: KaaS graph of a nearest-neighbors algorithm (equation 1). X_{inp} and A are read from the system data layer, while V_{iter} will be written back as an output. X_{tmp} and X_{iter} are ephemeral buffers that will never be copied off the GPU.

manage data transfers and scheduling. These assumptions are often baked into highly-tuned, but opaque, libraries. The KaaS model upends many of these assumptions. Additionally, placing GPUs behind a distributed abstraction will lead to additional invocation latency. Applications that finely interleave host and device code may not be able to amortize this additional latency.

The rise of serverless computing and modern machine learning frameworks suggests a promising path out of these challenges. Users have shown a willingness to decouple applications further in cloud applications, and backwards compatibility often takes a backseat to agility. The widely-adopted microservice architecture provides a good example of this [26]. These systems are a radical departure from traditional monolithic designs but enable rapid development, flexible deployment, and organizational flexibility. Another important trend is the rise of GPU-enabled application frameworks [62, 4, 203, 217]. Many GPU applications, and deep learning in particular, are no longer interacting with the GPU directly. Instead, these applications are increasingly relying on frameworks and libraries to generate GPU code transparently. In many cases, these frameworks already support, and optimize for, remote execution. KaaS can leverage these trends to make radical changes to accelerator interfaces with minimal impact on users.

5.2.3 Implementation

I now present an implementation of KaaS using the Ray distributed computing framework [181]. In addition to the core system, I provide a number of methods for users to implement kTasks, including a TVM-based deep learning compiler, a pre-made BLAS library based on Nvidia Cutlass, and hand-written applications.

Throughout this section, I will assume a scenario in which there are multiple *clients* submitting requests for a particular function. I use the term *function* to refer to a particular logical function, independent of any particular instantiation of that function. For simplicity, I will assume clients send requests to only one function and that different clients use different functions.

5.2.3.1 Ray Implementation

Ray Background Ray provides users with a Python API to run stateless functions called *tasks*, or stateful *actors* across a cluster. Data are communicated using references to objects in an immutable object store called Plasma. References are a form of *future* [24] and can be created before their associated object is available. Ray takes advantage of this to create lazily-executed graphs of tasks/actors that are scheduled only when their input references are available. Ray maintains a number of processes on each node called *workers* that execute tasks and actors. A single worker may execute many tasks, but actors are always run on a dedicated worker. To control resource usage, users may annotate tasks or actors with resource requirements such as the number of GPUs required. Ray then ensures they are run on a worker that has been allocated those resources.

GPU-Enabled Functions in Ray Ray does not enforce isolation of resources for tasks. Instead, it relies on applications to ensure they do not exceed their limits. Furthermore, tasks running on the same worker share resources at a fine granularity and must ensure that all resources are freed before exiting. This is difficult to ensure in practice, so the Ray documentation recommends forcing worker restarts on every invocation of GPU-enabled tasks [221].

Actors are persistent and run on dedicated workers. They are less sensitive to resource management concerns and can cache GPU state between invocations. However, there cannot be more GPU-enabled actors than available GPUs in the system and users must manually manage their actors to ensure that they do not exhaust available resources.

As a baseline, I enhance Ray with a new safe GPU-enabled task type called Exclusive Task (eTask). eTasks are written in Python in the same way as regular Ray actors and tasks. Unlike Ray native tasks, eTasks run on a dedicated worker per task with exclusive control of a GPU. They can opportunistically cache state between invocations. However, because eTasks have exclusive control of their GPU, the system may need to terminate them to free resources for new eTasks. eTasks provide an interface similar to many FaaS platforms like AWS Lambda or Google Cloud Functions.

KaaS Implementation in Ray To implement KaaS in Ray, I provide an additional GPU-typed function called a *kTask*. kTasks behave similarly to Ray's CPU-based tasks; they take in object references, output new references, and execute based on input availability. However, rather than providing Python source, kTasks are defined by a request object consisting of a DAG of kernels to invoke and buffer specifications describing input, temporary, and output buffers as well as literals for simple pass-by-value inputs.

kTasks are run by an alternative worker implementation called the *KaaS executor* (Figure 5.12). Much like regular workers, the KaaS executor is responsible for managing kTask code and caching objects from the object store. Unlike eTasks, the KaaS executor is responsible

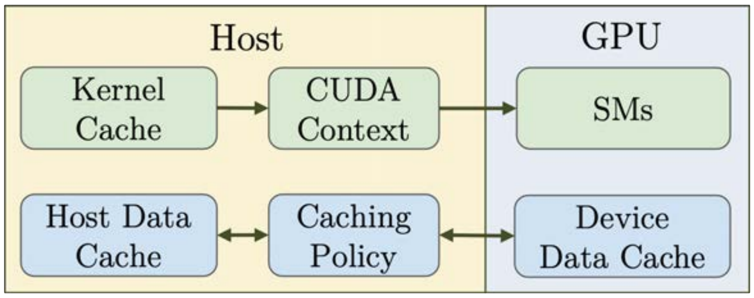


Figure 5.12: Design of the KaaS executor. CUDA kernel code and a data cache are maintained on the host. The GPU maintains its own cache, independent of the host, and runs user kernels to completion in a CUDA stream.

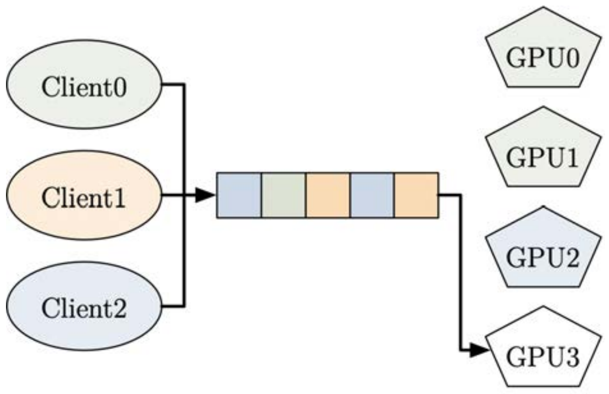
for managing the GPU rather than individual kTasks. This means that a single executor can handle any kTask without needing to restart. kTasks are routed to KaaS executors based on a centralized scheduler that augments Ray's internal scheduler to consider input availability, data locality, and per-GPU load.

Internally, the KaaS executor manages code through a kernel cache that handles linking CUDA libraries and preparing kernels for invocation. Upon invocation, the linked kernel is launched using regular CUDA APIs on a single stream. Currently, kernels are invoked serially, though future implementations could support concurrent invocation of non-dependent kernels.

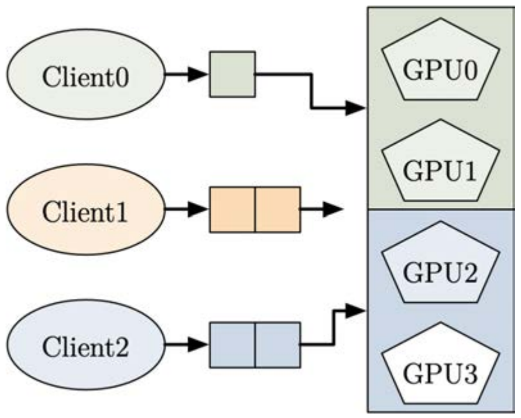
Data are managed through tiered host and GPU memory caches that extend Ray's built-in data layer. Objects are first loaded from Ray's object store into a data cache in host memory before being loaded into GPU memory. Ephemeral intermediate buffers are also cached in GPU memory to avoid frequent calls to CUDA's expensive memory allocator. The current design is a hybrid inclusive/exclusive cache where inputs are kept in both host and GPU caches, but outputs and intermediates exist only in the GPU cache. When GPU memory capacity is exceeded, the GPU cache first evicts from the set of objects with only one use before considering more frequently used objects. Both sets use a least-recently-used policy.

GPU Worker Pool Both kTasks and eTasks require custom workers to run on each GPU. For eTasks, these workers are user-defined actors. For kTasks, this is the KaaS executor. I implemented a custom worker pool mechanism in Ray to manage the scheduling of these workers. The worker pool includes two scheduling policies:

- **Balance:** The balance policy routes requests to the next available GPU regardless of locality or isolation. One permanent worker is started on each GPU at system boot time.



(a) **Balance Policy** Requests from any client can go to any GPU. The balance policy will route requests in FIFO order to the next available GPU. In this scenario, the request from Client1 will be routed to GPU3.



(b) **Exclusive Policy** There must be a client-specific worker on each GPU. Clients 0 and 2 have pools of GPUs assigned to them while Client 1 currently has no assigned GPUs. In this scenario, the exclusive policy would need to re-balance the pools, taking GPU3 from Client 2 and cold-starting a worker for Client 1 on it.

Figure 5.13: **KaaS Schedulers** In this scenario, clients are submitting a series of requests for a client-specific function. there are three independent clients and four GPUs. GPUs $0-2$ are currently running requests. GPU3 was previously running a request from Client 2 but is now idle.

- **Exclusive:** The exclusive policy ensures that no two functions run on the same worker, while repeated invocations of the same function are always routed to the same set of workers. To maintain this property, the exclusive policy must kill existing workers to make room for new requests if the number of unique functions exceeds the number of GPUs.

The Balance policy, depicted in Figure 5.13a, is straightforward. The policy creates a permanent generic worker on each GPU upon system initialization. Clients submit requests to a single queue on the scheduler. When a GPU becomes idle, the scheduler routes the next available request to that GPU. This algorithm ensures that no GPU is ever idle if there are pending requests. KaaS can safely use the Balance policy because the permanent worker is the KaaS executor which can service requests from multiple kTasks without needing to restart. eTasks require strict isolation between workers and cannot use this policy.

The Exclusive policy in Figure 5.13b is more complex. It must ensure that requests always run on a dedicated per-client worker rather than sharing a GPU execution environment. To do this, the Exclusive policy maintains independent per-client *pools* of workers. Internally, these pools follow the Balance policy, but they only service requests from one client. If a client submits a request that cannot be serviced from its pool immediately, the policy may consider shrinking an existing pool to free GPU resources for the new request. It begins by finding the largest pool to use as an eviction candidate p_{victim}. If there are multiple pools with the largest size, the least-recently evicted pool is chosen. If the requesting client's pool, p_{req}, is smaller than p_{victim}, the algorithm will evict a GPU from p_{victim} and assign it to a new worker in p_{req}. If p_{victim} has idle GPUs, they are simply re-assigned to p_{req}. Otherwise, the algorithm selects a busy GPU and waits for the currently executing request to finish before re-assigning the GPU to p_{req}. If p_{req} is in the set of largest pools, the algorithm simply blocks the request until a worker from p_{req} becomes available. I use the Exclusive policy for all eTask experiments.

5.2.3.2 Application Support

There are several ways to specify kTasks:

Low-Level API KaaS exposes a python API for describing kTask requests. Figure 5.14 depicts an example of a dot product request. These requests consist of a list of kernels to invoke and their associated inputs, outputs, and temporary buffers. Kernels may also take literal arguments that are passed by value in the KaaS request. The kernels themselves are described using a filesystem path to the compiled CUDA code and a kernel name within that file along with the appropriate CUDA grid and block dimensions to use when launching. In addition to kernels, I provide a simple control-flow mechanism for fixed-length iteration. These requests are then serialized by Ray and sent to the KaaS executor for processing. Callers immediately receive a reference to the future response, just like any other Ray task.

kTask inputs are passed as Ray object references, while output references are provided in the kTask return value. Internal buffers are only valid for the duration of the request and are not associated with the Ray object store. Upon completion of the request, any output buffers are written back to the Ray object store and a reference to this output is returned to the caller.

TVM-Based Deep Learning Compiler I worked with Anton Zabreyko to modify the TVM deep-learning compiler to generate KaaS-compatible code. While we chose TVM for our prototype, KaaS can behave as a backend to any deep learning framework that generates static graphs of CUDA kernels. At a high level, TVM functions by generating a linearized DAG of operations to execute the model. It then runs these operations sequentially, each of which consists of one or more CUDA kernels. This scheme has a 1-1 correspondence with KaaS graphs, with the only difference being that each operation needs to be expanded into one or more kernel nodes.

Our approach to converting the TVM runtime graph to a KaaS request is straightforward. First, we use TVM to generate the CUDA kernel library and the static runtime graph. We then extract the necessary information, such as the grid and block dimensions for each kernel. This information is then used to generate the KaaS code for the request.

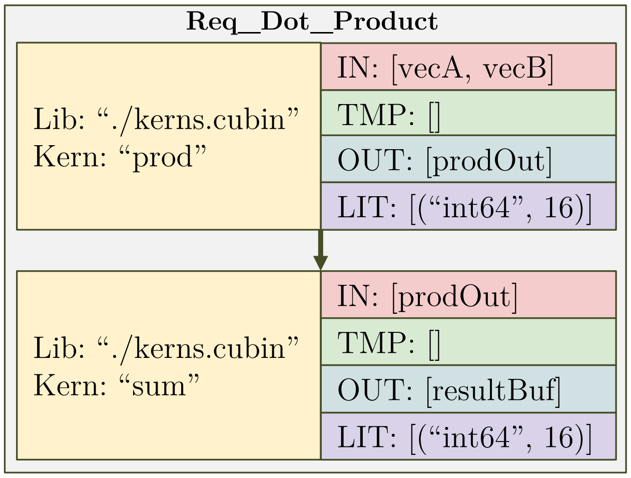


Figure 5.14: Low-level KaaS API request for a dot product. This request consists of two kernels (an element-wise product followed by a summation). The `vecA` and `vecB` buffers are loaded from the data layer while `prodOut` is an ephemeral intermediate result, it is only valid while the request is running. `resultBuf` will be written back to the data layer after the request has completed. Each kernel also takes a literal value representing the vector length.

BLAS Library The CUDA ecosystem has benefited greatly from high quality libraries of common kernels such as cuBLAS and cuDNN [71, 72]. These libraries are often authored by service providers or device manufacturers. Likewise, I expect a similar ecosystem would grow around a realistic KaaS deployment. For this project, Zhoujie Ding and I ported one such library based on Nvidia's Cutlass BLAS library [73]. Cutlass is a C++ template library that allows users to instantiate handles to specialized linear algebra kernels. We provide a KaaS interface to Cutlass through built in functions. Rather than uploading specific kernels, users can simply reference one of the system-provided functions. KaaS then manages interfacing with the library and managing device resources as needed.

While porting Cutlass to KaaS was non-trivial, it was a one-time cost borne by the system provider rather than individual users.

5.2.4 Evaluation

5.2.4.1 Experimental Setup

My experiments were performed on a single p3.8xlarge Amazon EC2 instance with 32 vCPUs and 4 Nvidia V100 GPUs (with 16 GB memory each).

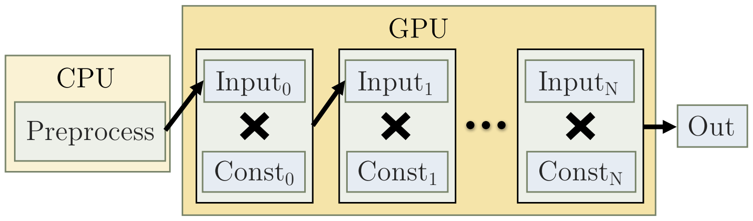


Figure 5.15: Chained matrix multiply micro-benchmark. Each multiply has one constant matrix and one dynamic matrix. Dynamic inputs are preprocessed before being passed to the first layer. Each subsequent step uses the output of the previous layer. The initial input, constants, and final output are stored in the data layer while the intermediate matrices are ephemeral.

5.2.4.2 Micro-Benchmarks

For this analysis, I will use the chained matrix multiplication micro-benchmark shown in Figure 5.15. Inputs are taken from the data layer, preprocessed on a CPU, fed through a series of multiplications on a GPU, and the output is written back to the data layer. Each multiplication uses a constant matrix that is read from the data layer but will not change between invocations (enabling caching). I configured the benchmark to use three layers with square matrices with a side length of 1024 single precision floats. The baseline eTask implementation required the numpy, pickle, and pycuda python modules. For cold starts, the system has not seen any requests for the benchmark. For eTasks, this means that it must create a new worker and initialize any python dependencies before running the eTask code. For kTasks, the system already has KaaS executors initialized since they are independent of any particular request. However, those executors must parse the request, link against the specified CUDA libraries, and load all data from the data layer. On warm starts, both task types already have a worker ready to execute the request immediately.

Figure 5.16 shows the results of this experiment. I begin by observing that kTasks and eTasks have similar warm-start performance (Figure 5.16a). This is not surprising as both implementations must perform the same steps. Any differences are mostly down to implementation details. Next, I move my attention to cold starts (Figure 5.16b). Here we see the drastic impact of starting a new Python process for eTasks. Even though my microbenchmark only imports a minimal set of packages, it still takes an additional 400 ms to load (a 46x increase). In contrast, the only cold-start overheads experienced by kTasks come from warming the data caches with the constant matrices. To see this in more detail, Figure 5.16c compares warm and cold starts experienced by kTasks. As expected, the time to load an additional three matrices adds significant time to both the data layer and GPU memory management phases, but there are no other overheads.

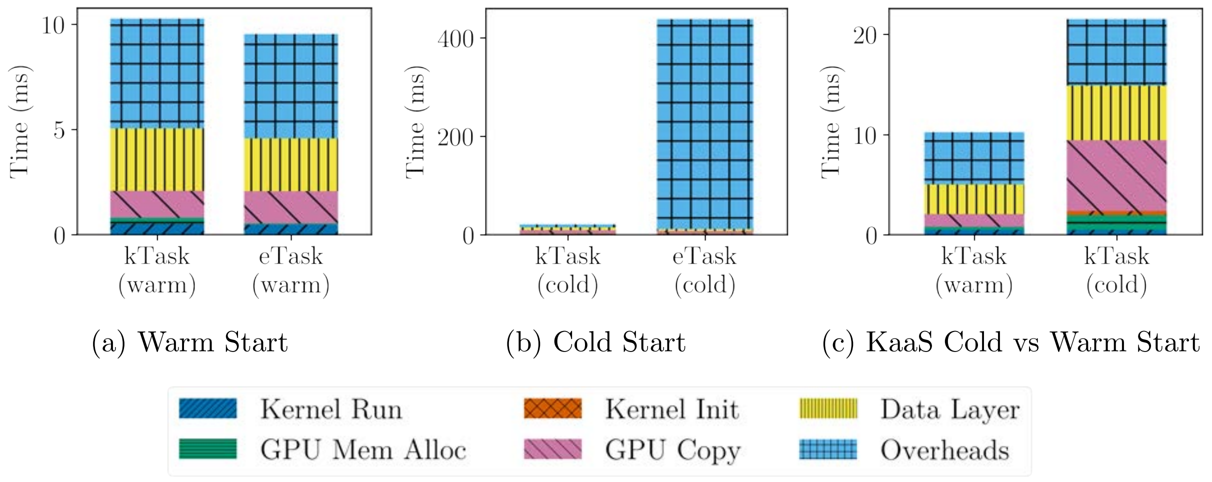


Figure 5.16: Microbenchmark Results. *Kernel Run* is the time spent actually executing the request's kernels. *Kernel Init* is the time to link any needed precompiled CUDA libraries. *GPU Mem Alloc* and *GPU Copy* encompass all device data management while *Data Layer* measures host data management. *Overheads* measure any additional tasks performed by the worker including request parsing, worker initialization, and Ray framework overheads.

5.2.4.3 Multitenant Workload

As we saw with the microbenchmarks, kTasks behave similarly to eTasks for a single warm workload. However, when there are multiple clients, the behavior changes dramatically. As discussed in §5.2.3.1, the eTask approach requires a policy that assigns GPUs to workloads exclusively, falling back to terminating and restarting workers as the number of workloads exceeds the available GPUs. These cold starts prevent the system from fully utilizing its GPU resources. KaaS, in contrast, supports a broader range of scheduling policies that can share a limited pool of GPUs more effectively.

I now evaluate four real-world applications in a multi-tenant environment with two scenarios: online and offline. In the online scenario, I evaluate how my system supports latency-sensitive workloads like model serving or interactive data analytics where requests arrive according to a Poisson process. To understand sustained throughput, I evaluate an offline scenario where all workloads submit requests as fast as possible. Two of my workloads, resnet50 and BERT, are deep-learning inference workloads generated by the TVM interface. These workloads consist of many small kernels operating over many small buffers. Deep learning models also have significant data re-use through the model weights. The cGEMM workload is a chained complex number matrix multiply using our Cutlass library interface. It multiplies a large constant 10000×25000 matrix by a narrow 100×10000 matrix that changes on each invocation, both contain 32 bit complex floats. This results in small inputs and outputs but a large amount of cacheable data. Finally, the Jacobi workload uses the low-level KaaS interface to implement an iterative solver using the Jacobi method [122].

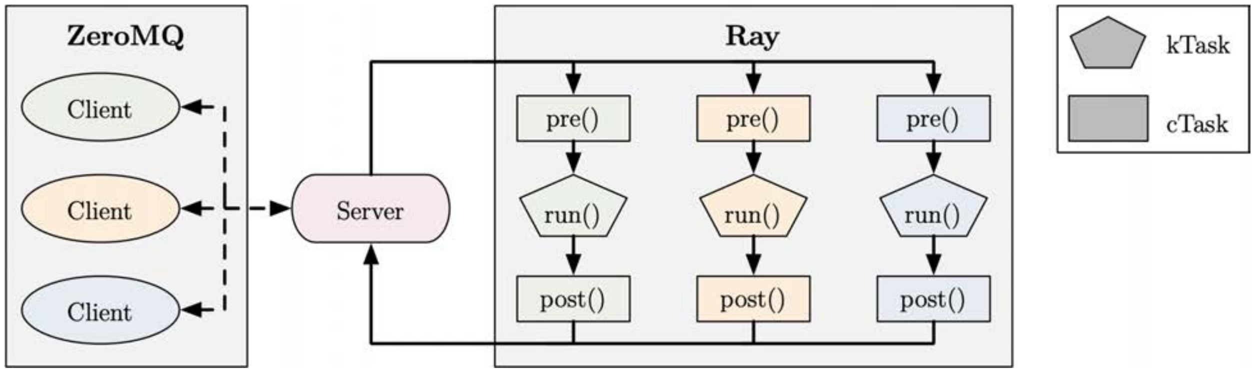


Figure 5.17: **Multitenant Environment** Clients submit requests over a ZeroMQ socket to a server that submits per-request function pipelines to Ray. Each request pipeline involves optional CPU-only pre and post-processing steps with a GPU-based execution phase. The server proxies pipeline outputs back to the client.

Name	Constant Mem	Dynamic Mem	GPU Time	Host Time
resnet50	129 MB	5.63 MB (TVM) 45.9 MB (KaaS)	3.8 ms	9.7 ms
BERT	1.34 GB	6.33 MB (TVM) 1.89 GB (KaaS)	91.8 ms	132.0 ms
cGEMM	2.00 GB	8.00 MB	38.8 ms	0 ms
Jacobi	0 MB	1.08 MB	52.1 ms	0 ms

Table 5.1: End to end workload properties. For the dynamic memory of resnet50 and BERT, TVM implements buffer recycling while KaaS does not, so we provide the two different numbers for this category. See §5.2.3.2 for more details.

This workload uses KaaS's iteration control-flow mechanism to run a fast update kernel for 3000 iterations. The input matrix contains 512×512 `float32s` and is accumulated into a 512×1 `float64` output, there is no data re-use between invocations. Table 5.1 summarizes the properties of these workloads.

In addition to the GPU-based functions, the workloads also include pre and post processing functions that occur exclusively on the host with no GPU requirements. These functions form a graph that is lazily executed as inputs become available. Finally, this experiment uses multiple external clients that communicate with the Ray-based service through a network protocol using ZeroMQ [302]. Figure 5.17 shows the complete setup.

Offline Workloads I begin with a throughput-oriented scenario where workloads from many clients submit requests as quickly as possible with no regard for latency. In this case, my key metric is the aggregate throughput of the system. This indicates whether or not the GPU resources remain highly utilized as the number of clients increases.

Figures 5.18 and 5.19 show the results from this experiment. We see that kTasks and eTasks both perform well when there are sufficient GPU resources to support the workloads. However, as I exceed the number of available GPUs, the eTask approach's throughput drops dramatically due to frequent worker cold starts. kTasks do not suffer significant slowdowns as I exceed available GPUs because the system is able to handle requests from different clients without restarting the KaaS executor.

The kTask implementation of BERT only begins to slow after 10 replicas. This is the point at which I begin to exceed available GPU memory for cached weights and KaaS must begin evicting and re-loading constant buffers. Unlike the sharp performance drop seen with eTasks, we see a more gradual performance impact with kTasks since the cache can be managed at a fine granularity. Even at 16 replicas, we still see significantly higher throughput (12 QPS for kTask vs 0.23 QPS for eTask). This comes down to the difference in cold-start behaviors between the two approaches. For eTasks, the system must boot a fresh python process and initialize any packages and state needed to run the model, while kTasks need only reload model weights from the host data cache. For BERT, the Python process takes 2.5 s to load while KaaS can load the model weights in only 160 ms.

Unlike BERT, resnet50 is a small model that fits easily within GPU memory. It experiences no significant slowdowns, even at 16 replicas. However, single-client warm-start performance is somewhat slower for kTasks than the eTask implementation. This is due to implementation differences between the two systems. The eTask implementation uses TVM's highly optimized C++ runtime while KaaS uses a simple Python implementation. While these differences are easily amortized by the larger kernels used by BERT or cGEMM, resnet50 consists of many very small kernels that accentuate system overheads. Fundamentally, the same work is done in both implementations and a more optimized KaaS executor implementation should behave similarly to TVM.

Finally, I note a slight dip in performance at three replicas for the eTask workloads. While KaaS is able to utilize any available GPU for any request, eTasks must be routed to the same actors each time. Re-balancing an eTask system would take several seconds. As a result, at three replicas one workload gets two GPUs assigned while the others get only one GPU. This load imbalance causes head-of-line blocking in our scheduler as the two workloads with only 1 GPU see significantly lower throughput than the two GPU workloads. While a more sophisticated scheduler may be able to alleviate this issue somewhat, poor load balancing is a fundamental property of the eTask approach.

Online Workloads In this experiment, I simulate a latency-sensitive online environment using the MLPerf Inference load generator in server mode [223]. Clients submit requests to the framework according to a Poisson process with their mean arrival rate set to 80% of peak throughput (to ensure stability). I report the median and 90th percentile response latency for each configuration.

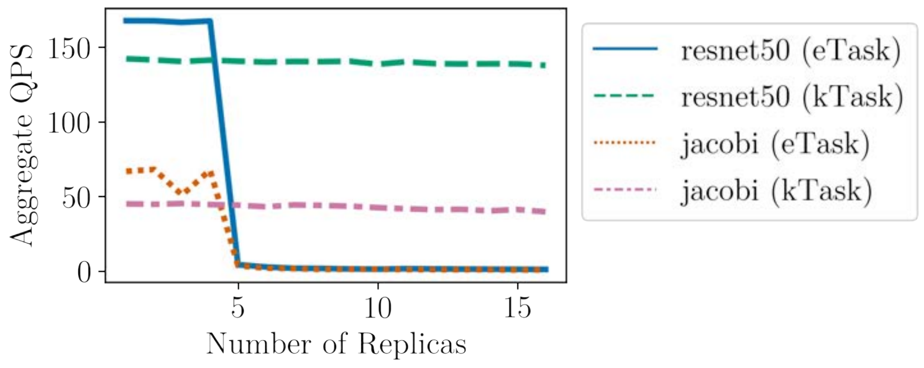


Figure 5.18: **Aggregate throughput of low-memory workloads** These workloads fit easily within GPU memory and see no aggregate throughput loss in KaaS as we increase the number of replicas. eTask workloads require exclusive access to GPUs and must begin cold-starting workers after 4 replicas.

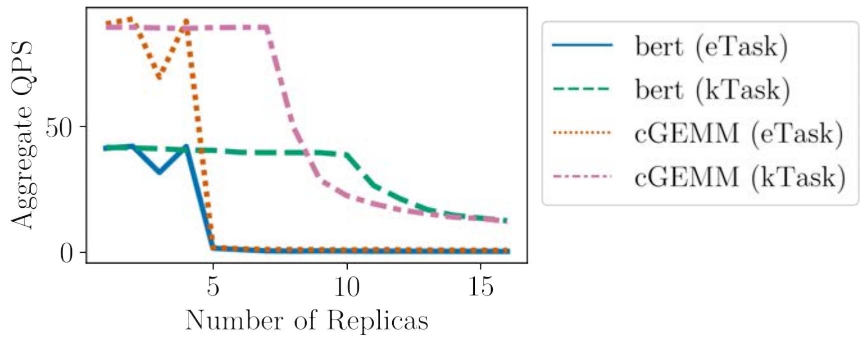
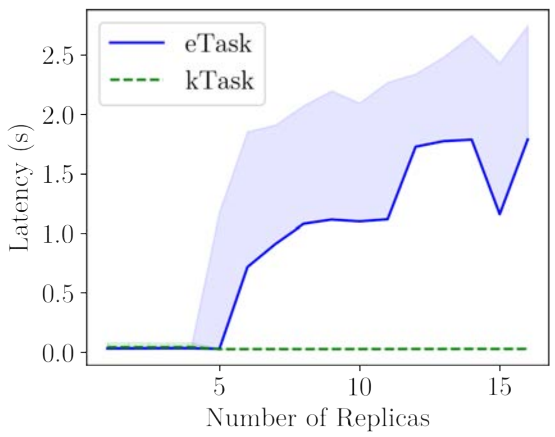
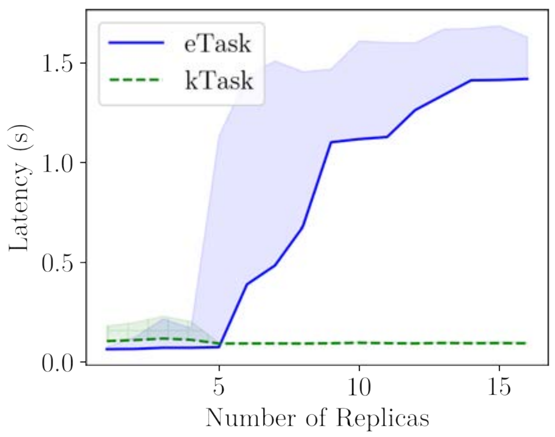


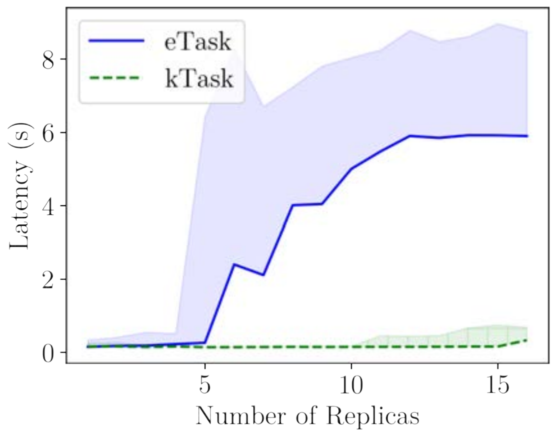
Figure 5.19: **Aggregate throughput of high-memory workloads** These workloads consume a large amount of GPU memory. When the aggregate memory requirements exceed available GPU memory, KaaS must begin evicting and re-loading objects from the GPU memory cache. This leads to a gradual decline in performance as cache pressure increases.



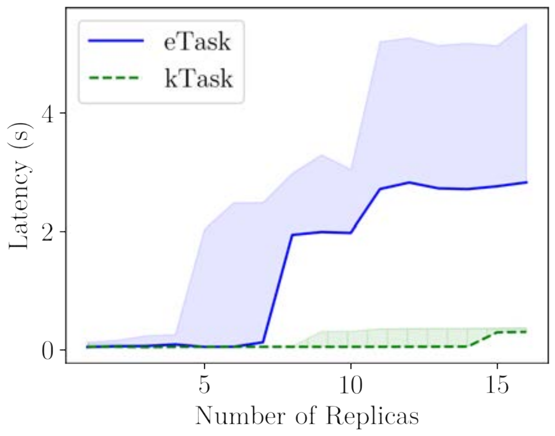
(a) resnet50



(b) Jacobi



(c) BERT



(d) cGEMM

Figure 5.20: **Online benchmark results** The median response latency is plotted along with the 90th percentile tail. For each model, we send requests under a Poisson distribution with the average arrival rate set to 80% of the slowest model's peak throughput.

Figure 5.20 shows the results of this experiment. We see very similar behavior to the offline scenario. When the number of clients does not exceed available resources, both approaches behave reasonably well. However, when I exceed available resources, the eTask approach is forced to cold-start new actors for nearly every request, resulting in very poor tail latency. In effect, the tail latency becomes a measure of cold start time. This latency depends primarily on the python dependencies and initialization tasks for each workload. kTasks are able to effectively share GPUs, resulting in no significant impact on tail latency, even as I exceed the number of available GPUs. However, the BERT kTask begins to slow down after 10 replicas as the aggregate constant memory exceeds GPU memory. Still, even at 16 replicas, the BERT kTask's tail latency is far below the eTask version due to the difference in cold start performance.

Figure 5.21 shows the CDFs of response latency at key points in the configuration space. Similar to Figure 5.20, I configure the experiment to submit requests at 80% of peak throughput to ensure queue stability. The maximum throughput of eTasks is significantly lower than kTasks so I plot CDFs for kTasks at both an equivalent submission rate to eTasks, and at peak kTask throughput. At 4 replicas, each replica has a dedicated GPU. In this case, both kTasks and eTasks behave similarly with only minor differences due to implementation details. For resnet50, the large number of small kernels highlights overheads in my Python implementation. BERT's kernels are large enough to effectively amortize these overheads while the KaaS executor's memory management and caching systems outperform eTasks on BERT's larger memory requirements. As I exceed 4 GPUs, the system must begin sharing GPUs between replicas. Even with only one extra replica, we see the eTask tail latency suffer significantly since replicas may need to evict an existing eTask before cold-starting. As I increase the concurrency to 16 replicas, even the best 10th percentile latency suffers as replicas become nearly guaranteed to require a cold start. In contrast, kTasks experience very consistent latencies as the number of replicas increase. Even at 16 replicas, there is no noticeable reduction in latency.

5.2.5 Other Approaches

The need to share and manage accelerators in a distributed system is not a new one. In §5.2.1 I covered several broad categories of techniques to achieve this. In this section, I briefly cover several specific approaches.

5.2.5.1 Single GPU Sharing

There are a number of techniques for sharing an individual GPU among multiple processes. Nvidia GPUs support multiple forms of sharing including CUDA contexts, MPS, or virtual GPUs [183, 194]. CUDA contexts allow sharing but do not enforce resource usage limits while MPS and virtual GPUs provide some level of resource limits. Techniques like rCUDA are similar to MPS or virtual GPUs but proxy requests across a network to a remote GPU [82, 128]. In all cases, individual clients are assigned a specific GPU resource and must man-

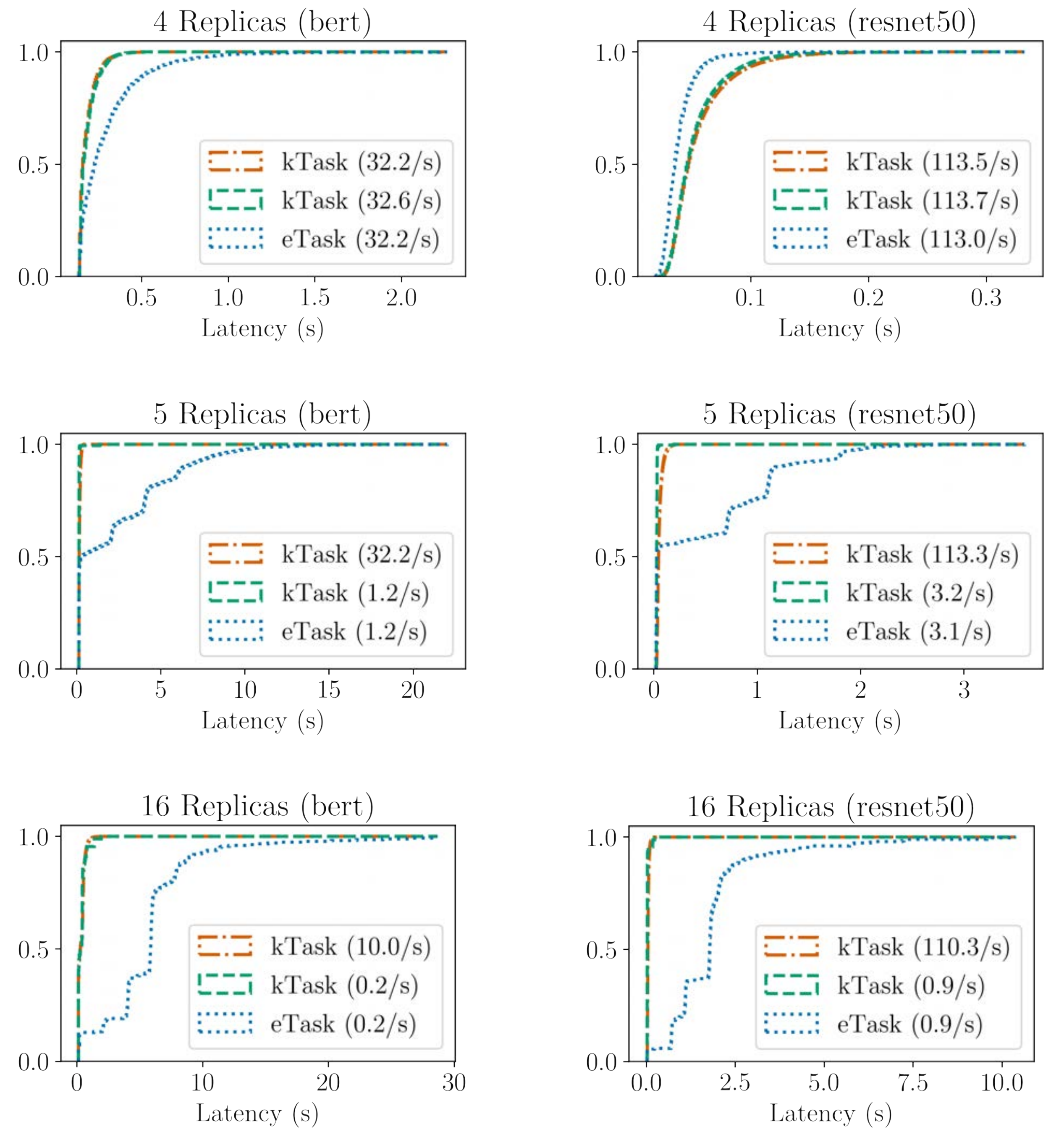


Figure 5.21: CDFs of BERT and resnet50 model response latency for different numbers of replicas.

age their own GPU resources explicitly for the lifetime of their allocation. While sharing increases the number of supported clients, those clients are still responsible for driving utilization of their allocation. They also do not address the issues of cold starts and support of general-purpose host code. KaaS allows for a more dynamic sharing of remote resources by abstracting all device management from clients, particularly data caching. Future versions of KaaS may take advantage of these sharing techniques to further improve device utilization, particularly for inter-request concurrency.

5.2.5.2 Domain Specific Systems

Some domain-specific services, particularly for deep learning applications, are able to share accelerators among multiple clients. Google Cloud's TPU interfaces decouple TPU devices from host servers and manage communication and allocation. However, applications are still assigned specific devices for their lifetime and are responsible for driving adequate utilization [261].

RAMMER is a deep learning compiler and associated runtime that improves device utilization by abstracting deep learning models and accelerators into finer grained components and optimizing their scheduling [162]. KaaS applications also provide graphs of CUDA kernels which could be further optimized by systems like RAMMER.

PipeSwitch [23] and Salus [299] are two frameworks to facilitate sharing of a GPU between multiple cooperating deep learning processes on a single node. Like KaaS, they abstract GPU resources by handling memory allocation and kernel scheduling for the clients. Unlike KaaS, PipeSwitch still allows direct GPU access from host applications and requires applications to cooperate. Salus is more similar to the KaaS executor because it accepts graphs of kernel requests and associated memory needs. Neither system integrates into a larger serverless environment, requiring a strong association between host process and server/GPU while KaaS allows for higher-level resource allocation/de-allocation. The design of the KaaS executor was influenced by these systems and would be improved by further application of their techniques.

There are also a number of systems specifically tailored to deep learning inference ([230, 105]) or training ([293]). These systems exploit the unique properties of deep learning, as well as cooperation from deep learning frameworks, to drive utilization and performance. Unlike these systems, KaaS is a general purpose serverless GPU interface for mutually unaware clients.

5.2.6 KaaS Conclusions

Serverless computing is a wonderful thing; it simplifies programming at scale, drives higher resource utilization, and frees users from complex provisioning decisions. As expensive application accelerators like GPUs rise in importance in modern workloads, these properties are more important than ever. Simply adding GPUs to existing serverless techniques will not be sufficient; they are too different. In this section, I presented KaaS, a truly serverless interface to GPUs that can integrate them naturally into the serverless ecosystem while preserving all the benefits we've come to know and love. KaaS frees users from tricky explicit allocations, effectively utilizes precious GPU resources in a multitenant environment, and frees system implementers to take full advantage of the unique properties of these devices.

5.3 Logical Disaggregation Takeaways

To reach the full potential of disaggregation, it's not enough to simply move resources around a network. We need a *logical* model of how users see and interact with those remote resources. In WabashOS, I started from the fundamentally aggregated logical abstraction of single-system image operating systems. I then enhanced that abstraction to be more flexible with resource allocations and distributed communication. Still, these abstractions needed to remain similar enough to traditional systems to support existing applications. This transparent disaggregation resulted in issues of noise that applications weren't designed to address. While gang scheduling techniques improved the situation somewhat, they sacrificed utilization to achieve familiarity. Unikernels made more radical changes to the user environment, but saw greater benefits in noise reduction without sacrificing utilization. Similarly, state-of-the-art techniques in distributed computing are *noise-tolerant* rather than noise-free. Again, these gains came because the logical model of execution deviated from familiar single-node abstractions.

In KaaS, I applied this insight to improving the performance and utilization of application accelerators like GPUs. Rather than requiring that users completely manage GPUs from a static server allocation, I changed the logical model to be more naturally disaggregated. This transition required users to adapt their applications to this new environment, but they saw huge benefits because of that effort. I assert that these two interfaces are not fundamentally more or less difficult to use, they are simply different. System interfaces like POSIX and the CUDA API were not designed for the multitenant and distributed systems we see today. In the final chapter of this dissertation, I will continue this argument by presenting a vision for a new, serverless, abstraction of WSCs that builds on the work I have presented so far.

Chapter 6

The Serverless Datacenter

In this chapter, I consider the implications of my work on future warehouse-scale computers (WSCs). I begin in §6.1 with a proposal for a new cloud system interface, while §6.2 suggests possible future research directions for serverless computing. §6.3 looks at the opportunities that such a system interface would enable for next-generation disaggregated hardware.

6.1 A POSIX for the Cloud

In Chapter 2, I described a number of existing interfaces to WSCs. These included *web services* that are based on static server allocations communicating over internet protocols like HTTP (REST). These static allocations led to idle resources and the coarse-grain servers could strand resources. They also struggled to take advantage of new hardware due to their stateless and highly-decoupled REST interfaces. There were also *distributed operating systems* that tried to make the WSC appear to users as a single node. These avoided stranded resources by transparently distributing work across the cluster, though that transparency came at a performance cost. They could also reduce idle resources somewhat by using more flexible process abstractions, but these resource allocations remained static and prone to idleness.

In Chapter 5, I argued that a new interface, called *serverless*, presented a more logically disaggregated abstraction that could address both idle and stranded resources. However, that abstraction remains incomplete. Indeed, my Kernel-as-a-Service (KaaS) project was needed to expand the model to heterogeneous compute resource types like GPUs by introducing the kernel task (kTask) function type. While that project made serverless GPUs practical, similar efforts will be needed to incorporate the ever expanding set of compute resources available to us. Existing serverless interfaces also say little about non-compute services like storage or WAN communication.

Today's clouds consist of a large set of constantly evolving services and interfaces that have grown organically over time. Contrast this with traditional single-node environments. While users have a wide choice of languages and frameworks, all applications follow the same fundamental patterns. Compute occurs through stateful processes while interactions with the outside world occur through a filesystem or other named read/write objects. The portable operating system interface (POSIX) arose to formalize these patterns, not just for portability as the name implies, but also as a model of how an operating system behaves [266]. The question now is: What should the POSIX for the cloud be?

This new "POSIX" needs to be flexible enough to enable rapid innovation in both hardware and software systems. It also needs to enable high performance and high utilization by naturally aligning the structure of applications with the physical realities in the datacenter. In this section, I present a vision, based on joint work with Johann Schleier-Smith [207], for a new approach that extends serverless computing into a complete system interface. I refer to this hypothetical interface as the *portable cloud system interface (PCSI)*. Any good interface will need abstractions for how computation occurs (§6.1.1), how state is named and accessed (§6.1.2), and how those abstractions interact beyond the boundaries of the PCSI (§6.1.3).

6.1.1 Compute Abstraction

I define computation as any transformation over state and refer to these transformations generically as *functions*. Functions receive state as input and produce state as output. They may also read and manipulate state as they execute. Figure 6.1 shows a hypothetical application structured around PCSI functions. These functions have three key properties:

- **Universal Compute Interface:** A function can be reimplemented without changing its external interface. Drop-in replacement is possible, even when the new function relies on new underlying technology. Multiple implementations of the same function can even be provided simultaneously, allowing an optimizer to choose dynamically among them to meet performance and cost goals [230].

- **No Implicit State:** Functions receive state, produce state, and interact with external state via the data abstraction, however they cannot rely on internal state beyond a single invocation. As with current serverless Function-as-a-Service (FaaS) offerings [56], or the vision of granular computing [148], this facilitates pay-per-use and allows functions to scale from a single invocation to thousands (or more).

- **Narrow and Heterogeneous Implementations:** A wide and evolving range of platforms may be used to implement functions. However, each function should focus on a narrow and resource homogeneous operation. This decoupling enables maximum innovation and helps resource allocation by isolating bottlenecks [202] and maximizing resource utilization.

Function arguments include explicit data layer inputs and outputs, and a small pass-by-value request body. Users store functions themselves as objects in the data layer, allowing them to be invoked by other functions. In addition to invoking individual functions, users can

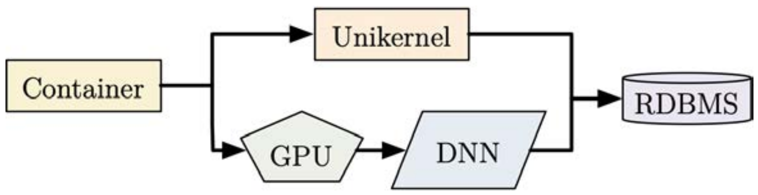


Figure 6.1: **Hetergeneous function graph in PCSI** While each function can be implemented in different ways, the general graph structure is common to all.

build task graphs, which opens up optimization opportunities such as pipelining or physical co-location. Such task graphs can either be specified ahead-of-time, as in Cloudburst [258], or dynamically, as in Ray [181] or Ciel [184].

PCSI functions are inspired by FaaS and share similar design motivations and aims. However, PCSI pushes these abstractions toward a more universal and integrated system interface. For example, rather than require distinct services for things like model serving or data analytics, PCSI exposes these features through the same interface as any other function. Likewise, new hardware and software platforms can be introduced without requiring new system interfaces.

6.1.2 State Abstraction

State in PCSI encompasses all information that is preserved beyond the lifetime of a single function. Access to state in PCSI is always explicit, which means that functions always access state over system interfaces. My design centers around a few key principles:

- **Universal Storage Interface:** Applications interact with state through a common interface. This ensures that the system has full visibility into communication and storage patterns, allowing it to optimize scheduling and placement for utilization or performance. This also provides a clear division between application and system, enabling implementations to evolve over time.

- **Everything is an Object:** If applications must use a common state interface, then that interface must be able to express the wide range of functionality available in the cloud. PCSI achieves this in much the same way as UNIX and its descendants [227, 212], by allowing various implementations of data layer objects. While some objects may represent persistent data, others may represent network connections or interfaces to system services.

- **Flexible Semantics:** There are many storage systems in today's cloud systems. These systems vary in their consistency, fault-tolerance, and security semantics, among others. Much as POSIX filesystems can have different properties, PCSI objects can have different semantics.

Though there is a diversity of object types and behaviors, PCSI insists on a common naming and reference system. Objects may be named, allowing identifier-based lookup. These names may be global or use a hierarchy similar to directories. Naming and persistence implies a level of coordination since the system cannot determine a priori where that object will be needed. For more ephemeral objects, functions can use unnamed *references* to objects. References allow the system to reason about object visibility and movement, enabling it to potentially elide costly serialization or consistency tasks.

As with functions, this state abstraction permits a great deal of heterogeneity. Objects may be stored in memory, disks, or application-specific data systems. In the case of ephemeral references, objects may even exist only within a function's memory. This abstraction also extends beyond traditional storage semantics by allowing objects to represent complex services like WAN connections. All this flexibility means that the system can choose to optimize for cost, utilization, or performance. It also means that new technologies can be integrated without deep changes to applications.

6.1.3 Fundamental Limits of Serverless

In system design, what is *not* included is just as important as what is. I believe that a serverless system interface like PCSI can enable a broad range of cloud workloads, but not every workload will run well as a collection of functions with explicit state. Things like transactional databases or highly synchronous stateful computations like those seen in scientific computing will always work best on explicit resource allocations. Still, these exceptions need not invalidate the larger system design. Instead, we can model these specialized computations as particularly large or expensive function types. While their resource allocations may be persistent, they can still invoke (and be invoked by) other standard functions. They may also read and write the data layer, or expose their services as specialized objects.

6.1.4 The Path to PCSI

While the system interface proposed here is far from complete, it provides the foundation for a truly disaggregated logical cloud interface. Indeed, I do not suggest that a system like PCSI would simply appear fully formed in the cloud ecosystem. Instead, cloud providers would evolve their serverless offerings over time to include more and more features. My work on KaaS represents one such incremental feature. Looking forward, that technique can be generalized to other resource types like FPGAs or deep-learning accelerators. Beyond KaaS, questions remain around appropriate consistency models, security, performance isolation, and many others. As these and other questions are answered, existing serverless offerings can evolve toward a common portable cloud system interface.

6.2 Frontiers in Serverless Computing

A core motivation throughout this thesis is that WSCs, and the cloud in particular, present an unprecedented opportunity for innovation. The challenge with this innovation is getting it into users' hands. Many of the existing interfaces described in Chapter 2 struggled with inappropriate abstractions that could not adapt to changing realities. Even my work on WabashOS was limited by a reliance on an aggregated process abstraction. Techniques like the PFA are able to integrate new technologies like remote memory, but remain fundamentally limited in their achievable performance. A flexible, fundamentally disaggregated, logical interface like PCSI offers the ability to quickly deploy new hardware or services. The KaaS model allows providers to deploy GPUs without affecting any other parts of the system. Similarly, the performance predictability of unikernels that I demonstrated in Chapter 5 came at the cost of generality. The heterogeneous function abstraction of PCSI would allow users to replace individual functions with this new high-performance OS environment without refactoring the entire application. There are similar opportunities for state. New memory or storage technologies become new object properties. New services or complex systems can be integrated as new object types.

In this section, I speculate on a few future directions in serverless computing and logically disaggregated interfaces.

6.2.1 Serverless Interfaces to Other Resources

The KaaS project focused on GPUs due to their ubiquity and central role in modern applications. How might we expand serverless to other resource types? FPGAs are an appealing target for disaggregation due to their high cost and ability to support diverse workloads. Microsoft has deployed FPGAs in their datacenters to support a wide range of workloads from deep learning to network function offload [215]. While these FPGAs are accessed through a remote function interface, functions are assigned to devices statically. One challenge with serverless FPGAs stems from their cold start behaviors. In addition to loading any constant data, FPGAs must be reconfigured to perform a new function. We must also define what an FPGA-typed function would look like. This includes a standard description of the FPGA configuration, standard interfaces, and resource accounting. In [225], Ringlien, et al., present a system architecture to address some of these concerns.

With the growth in demand for deep learning, cloud providers have begun to design compute accelerators with simple, application-specific, interfaces [127, 119]. These accelerators are typically programmed using a high-level framework such as Tensorflow or Onnxruntime [4, 195]. This high-level interface provides opportunities to simplify the serverless accelerator interface. Rather than requiring pre-compiled CUDA code, these interfaces could accept only high-level descriptions of the task. The provider would then be free to optimize for different metrics as needs arose. For example, functions could be compiled to minimize compute resource utilization to improve multitenancy or for maximum throughput. RAMMER describes one way that deep learning applications can be compiled for compute resource utilization in addition to raw performance [162]. High level interfaces also simplify security and isolation as the provider has full insight into application behavior. Deep neural networks have been shown to be very predictable, enabling the provider to make informed performance isolation and scheduling decisions [105].

6.2.2 Physical Co-Location

While the abstractions in PCSI are designed to support distributed systems, logical disaggregation does not imply physical disaggregation [258]. A näive implementation might send intermediate data from a CPU-based function to remote storage before pulling it onto a GPU. However, a more sophisticated implementation could use knowledge of application behavior to make much better decisions [39]. If the task graph indicates that the two functions will be composed, the system can schedule the first CPU function on a physical server that also contains a GPU. Since data were intended only for the next task, data movement is reduced to a single CPU-GPU memory copy. This implementation would achieve performance similar to a monolithic server-based interface.

While this approach improves performance, it also inherits challenges from monolithic designs. Direct communication between functions also couples their fates. If one function fails, the other is left in an inconsistent state and must also be terminated. We may choose to accept this situation; any function could fail regardless. We may also attempt to mitigate it using the approach proposed by MODC [138]. MODC uses disaggregated memory and idempotent semantics to provide fault tolerance to task graphs. Physical co-location also does not address the challenge of data serialization and deserialization. In monolithic approaches, all components of an application are carefully designed to support a common data format. In Serverless systems, we prefer to decouple applications to enable agility, much as was done for microservices [26]. However, portable protocols and serialization formats can be expensive. Google reports that roughly 5 % of fleetwide cycles go to protocol and serialization processing [130]. Future designs that use physical co-location or memory-semantic access to disaggregated memory will need to carefully design their data formats or use hardware acceleration to mitigate serialization impacts [133].

6.2.3 Scheduling for Diverse Metrics

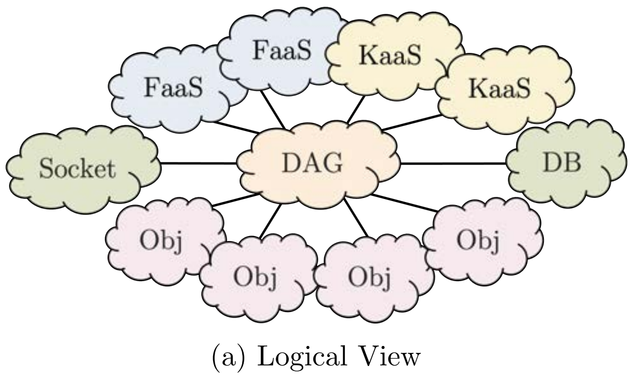
Performance is not the only metric of concern in cloud computing. Indeed, the work I presented in this dissertation has primarily been focused on improving utilization, even at the cost of performance. Logically disaggregated systems like PCSI also enable the flexibility to optimize for a range of metrics. While a physically aggregated strategy would provide the highest raw performance, the system may instead choose to utilize an idle resource from elsewhere in the system to avoid stranding local resources. If there are multiple implementations of the same function, the system may choose the cheapest implementation based on current system state [230]. In the case of applications with service-level objectives (SLOs), lower performance will have little impact on user satisfaction so long as the SLO is met.

As discussed in Chapter 5, FaaS systems must decide between expending resources to keep a worker warm or terminate it and run the risk of a cold start [245]. While KaaS does not assign workers to functions exclusively, it still must consider the resource usage/warmness tradeoff. In the case of KaaS, warmness is determined by the presence of a kTask's inputs in GPU memory. When a kTask is sent to a GPU, it may displace data from other kTasks. However, strictly assigning kTasks to GPU pools might prevent the system from utilizing idle GPUs outside the pool. Furthermore, we may encounter situations where all warm GPUs are busy. It may be the case that using a cold GPU would have lower latency than waiting for a warm GPU to complete its current request. Clockwork presents a method for making this determination [105], but focuses on the context of a single-application model serving environment. As we continue to expand the scope of serverless resources, we will need to design efficient algorithms to trade between utilization, throughput, and tail latency. Aditya Ramkumar's masters thesis presents this topic in more detail [218].

6.3 Logical Enables Physical

A core argument of this dissertation is that physical and logical structures are fundamentally intertwined. An inappropriate logical model fundamentally limits the achievable performance of new hardware technologies. Hardware features and limitations become assumptions built into our logical interfaces. When these two perspectives are not aligned, systems fail to reach their full potential. WabashOS's logically aggregated model was limited in its ability to improve utilization. New network and memory technologies promised disaggregation, but transparent logical interfaces led to poor performance. This is not to say there is no opportunity for advancement. Indeed, the page fault accelerator (PFA) saw performance improvements over a pure-software approach to remote memory. Similarly, my evaluation of KaaS in Chapter 5 showed benefits in performance and utilization even with today's hardware. In this section, I argue that further alignment of the physical and logical models will enable far more progress on the goals of utilization, cost, and performance.

In the previous section, I proposed a truly disaggregated logical interface. Figure 6.2 compares this interface to the physically disaggregated datacenter proposals of Chapter 2. There is a deep symmetry between these two approaches. New fast networks enable tightly coupled application graphs. Rather than the implicit interfaces of the PFA, physically remote memory is naturally aligned with an object abstraction [137, 6]. Where serverful approaches would need to create new server types for a new device, KaaS-like approaches enable standalone deployment of new devices. In §6.3.1, I explore how accelerators could be designed from scratch to support a KaaS-like interface while §6.3.2 describes how notions of locality can be applied to a disaggregated setting.



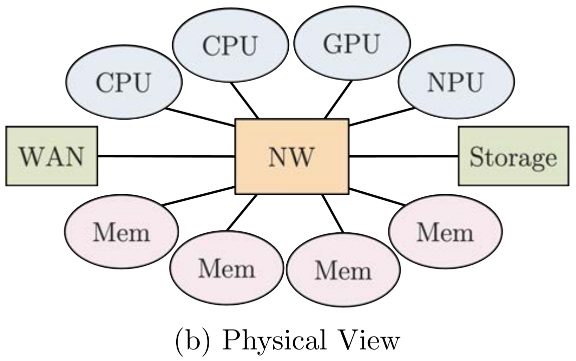


Figure 6.2: Two perspectives on a disaggregated datacenter. A serverless logical abstraction mirrors the physical organization of DDCs.

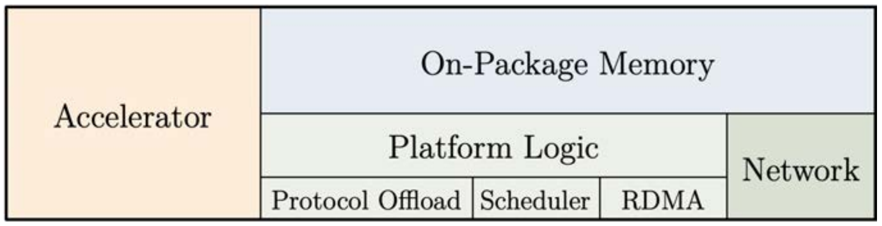


Figure 6.3: A dedicated serverless system on chip (SoC) would package only the bare minimum hardware to support KaaS-style functions. Due to the constrained workload, platform functions like protocol handling, serialization, or RDMA can be offloaded to custom logic.

6.3.1 A Serverless SoC

In Chapter 5, I presented systems that took advantage of the unique properties of existing devices to improve utilization. In the process, I created interfaces with much simpler semantics and much more predictable requirements. KaaS executors do not need to run arbitrary host code and they do not need a full-featured operating system. They also do not present a particularly challenging CPU workload to the host. The vast majority of time is spent either on the accelerator itself or in handling predictable network and protocol tasks.

As applications move toward a PCSI model of computation, cloud providers will become incentivized to optimize their hardware platforms for these properties. Figure 6.3 shows a hypothetical *serverless SoC* that contains only the bare-minimum hardware needed to support KaaS-like workloads. Network connectivity could be provided by high performance integrated silicon photonic networks [262]. Common platform tasks would be supported through either specialized logic [133, 147] or through a low-power control CPU. The bulk of package area would be consumed by the application accelerator and local high-performance memories.

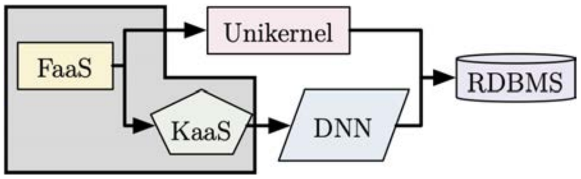
These specialized SoCs bring a number of important benefits. First, the platform accelerators and control CPU can be sized to handle the serverless protocol and no more. This will save significant power over a high performance general purpose processor. Since the provider controls the entire platform, they can quickly iterate on designs. Deploying these devices is also simplified. Adding additional accelerator capacity is as simple as adding a new card to the network. There is no need for additional server hardware as is the case for traditional PCI-E attached accelerators. Physical racks, power supplies, and networks can all be tailored for these fine-grained SoCs. These benefits have already been realized in the storage space [91, 25]. Microsoft's Catapult system and Google's TPU pods similarly demonstrate the potential benefits of resource-specific hardware platforms [261, 215].

6.3.2 Locality Under Disaggregation

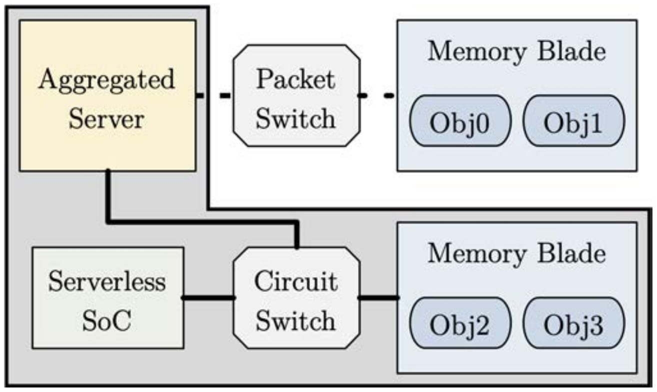
As discussed in §6.2.2, the requirement for explicit state may lead to additional data movement and serialization costs. The fine-grained decomposition of functions may lead to additional synchronization and complicates fault tolerance. Fortunately, the *ability* to disaggregate does not imply the *need* to disaggregate. Figure 6.4 shows both a logical and physical perspective on how this fact might be exploited. While users provide a disaggregated graph of functions, they may additionally indicate that subgraphs of their application are likely to be tightly connected. In this case, we may choose to *fate-share* within that subgraph. This allows us to elide expensive synchronization or fault-tolerance steps by simply failing the entire subgraph if any part of it fails.

Physically, we may exploit knowledge of tightly-coupled subgraphs to co-locate functions on the same physical server or under the same top-of-rack network switch. As datacenter hardware becomes more specialized for the serverless setting, locality becomes even more flexible. A serverless SoC, coupled with a high fanout photonic link, can use high performance optical circuit switches to create *pseudo-physical servers* where every component is directly connected over high bandwidth, low-latency photonic links. In some cases, we may want to directly connect two compute resources to avoid data materialization. In other cases, we can exploit reconfigurable locality to ease data movement, exploiting the duality of memory and communication [298]. In a physically disaggregated setting like Figure 6.4b, memory resources can be directly connected to compute resources. In this example, Objects 2 and 3 are directly available to the compute resources while Objects 0 and 1 must be addressed through a higher latency packet switched network. If a new function requires a serverless SoC and Object 0, the system would reconfigure the network to ensure maximum locality.

As with physical co-location, network reconfigurability provides a new capability, but not a new requirement. The system must decide whether to move data explicitly or reconfigure the network. It must also decide the optimal connectivity for each task in a constantly changing environment. The right policies will require new research into serverless task scheduling and placement.



(a) **Logical Optimizations** The two functions can elide consistency protocols and serialization if they are known to communicate directly and are allowed to fail together.



(b) **Physical Optimizations** The physical resources needed for the application sub-graph can be chosen from physically adjacent nodes. New environments with circuit switching and low-level physical disaggregation can create *pseudo-physical servers* with temporary tight connectivity.

Figure 6.4: Application insights can enable opportunistic co-location and optimization. In this example, the FaaS function is known to have significant communication with the KaaS function. These functions are instantiated on a traditional aggregated server and a specialized serverless SoC, respectively.

6.4 Concluding Thoughts

WSCs have given us an unprecedented opportunity to invest in radically different hardware and software systems. The work I presented in this dissertation has shown some examples of how we can take advantage of that opportunity to improve utilization and performance through disaggregation. However, many more challenges remain. As we look toward the future, new system interfaces will be needed to provide a framework for future innovation in disaggregation. I propose one such approach through the PCSI abstraction. With this abstraction, we can expand the serverless abstraction to new resource types like FPGAs or deep learning accelerators. We can even embrace the serverless model when designing new datacenter hardware like accelerator SoCs or network technologies. As always, the physical and the logical will need to be designed to work together to reach their full potential.

Chapter 7

Conclusion

I end this dissertation with a summary of the key lessons to be learned from my work (§7.1). I also include some reflections in, §7.2, on the practice of research and the lessons I have learned throughout my PhD journey.

7.1 Lessons on Disaggregation

In this dissertation, I have described a wide range of approaches to providing high utilization with reasonable performance through disaggregation. In the process, I presented methodologies for hardware/software co-design, techniques for physical disaggregation of memory, and logically disaggregated interfaces. In this section, I summarize my findings from these projects with an eye toward the key lessons to be learned from them.

7.1.1 Hardware/Software Co-Design Methodologies

My work explores how hardware and software can work together to improve the performance and usability of resource disaggregation. As we saw with the PFA, it is not enough to evaluate just the software or just the hardware; they need to be *co-designed*. This process is not trivial. Prior techniques were not sufficiently agile for a small team to evaluate the wide range of possible designs. Hardware and software designers were too decoupled by disparate simulation technologies and ad-hoc software workload management. Disaggregation made this even more difficult as existing evaluation methodologies struggled to support clusters of heterogeneous resources and complex end-to-end software.

The solution was to build agile hardware design methodologies. In my work, I focused on managing software workloads in a way that was reproducible, repeatable, and reusable by the community. I did not just consider the process of building static software images, but instead focused on the complete *software workload life cycle*. This included *building* the software images, *launching* them in functional simulation, *installing* them to external simulators, and repeating this process for *testing.* This broadened definition led to FireMarshal, my tool for managing this life cycle.

Chipyard combined software workload management with tools for hardware design, simulation, and manufacture into a complete SoC generation framework. This framework formed the basis of my methodology for designing and evaluating hardware/software solutions to disaggregation. Rather than relying on microbenchmarks and hardware models, the PFA project used real, synthesizable, hardware designs with end-to-end benchmarks running on a real operating system. This process exposed realistic behaviors and limitations that may have been masked by a less rigorous approach.

My methodology goes beyond just a single project. FireMarshal provided an unambiguous description of my software workloads. Others in the community are free to use those descriptions to reproduce my results, or as a basis for new research. Tools that build on common designs, share their outputs openly, and automate the process of experimentation can accelerate the pace of progress in our community.

7.1.2 Physical Disaggregation

Physical disaggregation can take many forms. I identified two important axes in this design space: implicit/explicit, and hardware/software. Paging to remote memory took an implicit approach by automatically moving data between local and remote memory, but existing approaches were hampered by slow software management. The PFA added hardware support to improve performance, but did not fundamentally change the implicit interface. While performance did improve, paging still required significant work from the OS. This limited the achievable gains. Nephele used an explicit interface to disaggregated memory to improve the performance of process checkpointing. Users had to change their applications, but they saw large performance gains as a result.

When physically disaggregating memory, we need to consider both axes. Hardware acceleration can improve performance, but it is costly and may not always be feasible. Implicit interfaces are easy to deploy, but explicit interfaces have the potential to be much faster.

7.1.3 Logical Disaggregation

The two axes of disaggregation design are fundamentally intertwined. An explicit interface may enable a richer set of hardware features, while implicit interfaces constrain our options. Some interfaces may only be feasible with certain hardware capabilities. To get the most out of disaggregation, we need to consider the *logical* view of the system that we present to users.

Web services and virtual machines focus on aggregated servers and long-running allocations. This leads to both stranded and idle resources. It also assumes too little about the hardware capabilities of modern datacenters. Distributed OSs embrace the tightly-coupled nature of WSCs, but they offer an implicit interface that limits our ability to drive utilization and performance. In WabashOS, applications designed for a single node suffered from poor performance predictability on a disaggregated system. My attempts to improve that predictability sacrificed utilization for performance. However, my work on KaaS showed how a fundamentally disaggregated interface like serverless computing could improve both performance and utilization. As WSCs evolve, we must ensure that their logical interfaces align with their needs and capabilities.

7.2 Reflections on Research Practice

I would like to finish this dissertation with some reflections on the practice of research. In §7.2.1, I opine on the drawbacks of writing high quality software for research while §7.2.2 reflects on the lessons I learned about managing research projects.

7.2.1 Good Software Considered Harmful

The fantasy author Terry Pratchett once wrote of his "boots theory". It goes like this:

> *A man who could afford fifty dollars had a pair of boots that'd still be keeping his feet dry in ten years' time, while a poor man who could only afford cheap boots would have spent a hundred dollars on boots in the same time and would still have wet feet.* – Terry Pratchett

This also holds in the world of software development. An initial investment in software engineering and testing can pay dividends when it comes time to extend or debug a codebase. However, there is an alternative theory that I will call the *wrench theory*:

> *I once needed a special wrench to remove an odd component on my car. I could buy one from the local discount store for one dollar, or order a twenty dollar wrench from a reputable dealer. If the cheap one works once, I have saved nineteen dollars. If it breaks, I have only lost one dollar.* – Nathan Pemberton

The moral is, the first time you buy something, buy cheap. The second time you buy it, buy expensive. In research, we often want to quickly prove or disprove a hypothesis. In all likelihood, our code will be abandoned once the paper is published. In the unlikely scenario that the project sees adoption in the real world, it can be re-written correctly. This is not to say that the code should be unrealistic or incorrect; it should be as good as necessary to demonstrate the problem and solution, but no more.

As counter-intuitive as it sounds, "good" software can be harmful to pure research projects. Rather than quickly proving hypotheses or exploring design spaces, quality software plans for scenarios that may never come to pass. What good is extensibility when the experimental code will be thrown out after the paper is published? Is it worth spending 25 % of your time on testing for a codebase that will never run production workloads [61]?

I have seen this in my own work. Johann Schlieir-Smith and I attempted to develop a flexible platform for serverless research called the Serverless Research Kit (SRK). The goal was to create a unifying environment for the broad range of serverless research happening at UC Berkeley. SRK included extensive testing, good documentation, and an elegantly modular design. It was an expensive wrench. Despite our best efforts, the framework saw little adoption. In practice, it was often easier to hack something together quickly rather than extend SRK to meet some new requirement. As bugs and incompatibilities arose in off-the-shelf software, we simply worked around them rather than fixing them. SRK did not provide enough value to justify the time investment.

When I began the KaaS project, I made a conscious effort to write "bad" code (the cheap wrench). This meant that new features often introduced bugs that were only caught much later due to insufficient testing. Poor software architecture and a lack of documentation meant that new collaborators were difficult to on-board and new features often required significant re-factoring. Despite this, the KaaS project moved quickly, producing publishable results in under one year. While not suitable for practical use, the current codebase and subsequent paper are more than enough to guide a serious implementation effort if the need arises.

Of course, the cheap wrench sometimes breaks. My early work on the PFA was plagued by frequent mistakes and setbacks due to poor software workload management practices. Entire experiments were lost due to accidentally deleted code. Work done several months earlier was unlikely to still work, if I even remembered how to use it. I also spent a large amount of time and money debugging software in RTL simulation due to the difficulty of switching between simulators. These problems were being faced across the research community. It was time to buy the expensive wrench. FireMarshal required a significant time investment, but has drastically eased the co-design process for my own work and the work of others. Indeed, FireMarshal arguably constitutes my most visible and influential work, being used by over 20 institutions.

The definition of "bad" is a tricky one. Often, new research directions appear while trying to solve seemingly boring problems. In other cases, buggy code may mask real effects that should be reported. Even bad code must be able to run real benchmarks in realistic settings. Researchers also must be careful to inspect results and make sure all data makes sense. In the KaaS project, I noticed unrealistically poor performance when running concurrent workloads. While I could simply have published those results. I instead investigated the discrepancy. It turned out that the load generators at each client were using the same random seed. This violated the independence assumption of queuing theory and led to poor performance. In this case, my cheap wrench broke. This was not a failure of the wrench theory; many other cheap components worked perfectly well. To quote an old Russian proverb: *"trust, but verify"*.

7.2.2 What to Work on and When to Give Up

7.2.2.1 Picking a Problem

In research, picking a problem is often harder than finding a solution. A good problem must be well motivated and have a potentially interesting solution. It must also be solvable within a reasonable time frame. A more subtle requirement is that it must have a context within your research group. A team can provide diverse perspectives on a problem, supply benchmarks, and has the ability to leverage each other's work. This is especially true in systems and hardware projects that require a large amount of engineering. Often, many distinct problems must be solved to build a realistic system. Without a team to support you, projects go slowly and have poorer outcomes. You may have to adjust your research focus to find a good team, but the benefits are well worth it.

7.2.2.2 Identifying Negative Results

Once you have started a project, the next decision you must make is when to stop. Research is, by definition, uncertain. Some ideas simply don't work. Other times, a problem you thought was solvable in a reasonable time frame is in fact intractable. These are called *negative results*.

When encountering a negative result, it can be difficult to differentiate between the normal challenges faced by all research projects and more fundamental problems. This is made particularly difficult in an environment rife with imposter syndrome. From the outside, other students and projects can seem so successful as to set an unrealistically high bar. In my own experience, I often abandoned projects too soon because their results seemed inadequate and their problems insurmountable. In hindsight, many of these projects provided significant contributions to the community despite their problems.

The PFA saw only 20 % improvements on many benchmarks which seemed insignificant at the time. However, paging techniques have seen interest despite their limitations [104, 52, 171]. Techniques like the PFA would have significant impact on those projects. Furthermore, the PFA would require changes to application processors to see real adoption. At the time, this seemed unlikely as cloud providers overwhelmingly used commodity platforms. Today, there has been great interest in developing fully custom SoCs and hardware platforms for cloud settings [156, 31]. It is hard to predict how a project may influence or be influenced by developments in the future. I saw this in the Nephele project as well. While Nephele showed large performance improvements, I was concerned with the difficulty of adapting applications to a new interface. The rise of serverless computing and cloud-native applications has shown that users are, in fact, willing to make deep changes for large gains.

7.2.2.3 Dealing with Negative Results

If a project truly has deep problems, you have to decide what to do next. In some cases, the project is simply not worth pursuing further. In these cases, it is useful to write a detailed postmortem of the project. In this paper, you articulate in detail why the project cannot proceed and what insights you have learned about the underlying problem space. Even though the academic community has a publication bias against negative results [86, 42, 213], these types of analyses often lead to new, more productive, projects.

Often, projects fail not due to the proposed solution, but due to limitations of the problem statement or evaluation methodology. In the PFA project, I focused on relatively small benchmarks. These benchmarks ran a specific algorithm against fixed input data from start to finish. This is a particularly poor fit for resource disaggregation as they are likely to use all of their memory. I also configured my benchmark to measure memory pressure based on the actual memory usage of the benchmark rather than *requested* memory. As previous research has shown, users often overestimate their memory requirements [66, 268, 176]. If I had compared against business and big-data applications, memory disaggregation may have looked more appealing.

In these situations, it is important to remain humble. As one person, you have limited perspective. When a project is struggling, it is important to seek the advice of others in your community. The first step is to consult with your immediate colleagues. After that, it can be worthwhile to submit your project to a conference or journal, even if you are unsure of the impact of your results. If your methodology is sound, reviewers may find the results compelling. Even if they don't, they can provide valuable insights into how you should proceed. It is arrogant to be convinced of your own greatness, but equally arrogant to be convinced of your failure. Be humble.

7.3 Departing Thoughts

We have the ability to design amazing hardware. Networks are approaching sub-microsecond latencies and Tbit/s bandwidths. Memory, storage, and compute technologies are all advancing rapidly. These *physical* advances provide benefits to today's computers, but unlocking their true potential will require re-thinking the *logical* model that we present to users. In this dissertation, I have applied this principle to one key metric of today's warehouse-scale computers: resource utilization. I built systems like the PFA and WabashOS that provided physical disaggregation on top of familiar logically aggregated interfaces. While these projects enabled higher utilization, their performance was limited. When I changed the logical interface to be more fundamentally disaggregated with Nephele and KaaS, I saw far greater gains in both utilization and performance. As we look to the future, we will need new unifying logical models like PCSI that will bring the advantages of disaggregation to a broader set of workloads. Hardware will also need to advance, and I have shown how agile hardware/software co-design with FireMarshal and Chipyard enables that advancement. Together, these techniques will enable a truly serverless datacenter that utilizes its resources effectively, provides high performance, and enables future innovations in hardware and software.

Glossary

anonymous page A page that does not contain disk-backed information. This is primarily "heap" memory (e.g. memory allocated through malloc()) 55

application programming interface A standard set of user-controllable features that a system supports 8–11, 15, 18, 66, 86, 87, 92, 93, 95, 96, 106, 126

cgroup The per-task (or group of tasks) resource management system in the Linux kernel. 55

Chipyard A system-on-chip development framework developed at UC Berkeley. 6

CPU task Traditional CPU-oriented serverless functions 88, 126

CUDA NVidia's GPU programming language 85, 88–90, 93, 95–98, 103, 105

disaggregated datacenter A warehouse-scale computer design that physically deploys individual resources as globally accessible network-attached components. 12, 126

EvictQ Queue of pages to be evicted by the PFA. Populated by the OS when it needs to free local physical memory. 50, 51

Exclusive Task A FaaS+GPU approach where serverless functions run with exclusive access to a GPU. 92–95, 97, 98, 100, 101, 103, 126

FaaS Function-as-a-Service: A programming style focused on explicit state and ephemeral functions that operate on that state. 4, 11, 85, 86, 89, 92, 108, 109, 113, 116

frame Synonym for page frame *Glossary:* page frame

FreeQ Queue of free frames to be used by the PFA to service page-faults. 50, 52, 53, 58

function A logical transformation over data that can be executed by a serverless framework. Functions can be implemented in many ways. 91, 92, 94, 99

kernel task The GPU-specialized serverless function type proposed by KaaS 86, 88–95, 97, 98, 100, 103, 107, 113, 126

Kernel-as-a-Service My proposed GPU-specific extension to the FaaS paradigm. 85, 86, 88–101, 103, 105–107, 110, 111, 113, 114, 119, 120, 122, 126

least recently used An algorithm that attempt to pick pages that have not been used recently. 55, 57, 126

network interface card The hardware device providing network connectivity. This term is used even if the network device is not implemented as a discrete external card. 41, 51–53, 60, 61, 126

NewQ Queue of new-page descriptors populated by the PFA on every page fault and drained by the OS for bookkeeping. 50

non-uniform memory access A system where memory is cache-coherently available to multiple CPUS, but with varying access latencies and bandwidths (a type of multi-socket machine). 43, 44, 126

non-volatile memory Storage devices with near-DRAM performance, and byte address-ablity, that do not lose their data when powered off. 44, 126

page fault accelerator The proposed hardware-accelerator that handles page-faults for remote pages automatically. 29–32, 47, 49–53, 55, 57–62, 64, 65, 113, 117, 118, 120–122, 126

page table A hardware-visible tree in main memory that contains translations from virtual to physical addresses. 57

page table entry A single entry of the page-table. Each PTE refers to a single virtual page. 50–53, 56–58, 61, 126

pageID A unique identifier for a page in remote memory. Acts as a remote-memory address. 50

partitioned global address space A language-based technique that partitions the program's address space between local and remote objects. 45, 126

portable cloud system interface A hypothetical unified system interface to warehouse-scale computers based on serverless computing. 108–112, 114, 116, 122, 126

remote direct memory access A system where memory is directly addressable between multiple nodes through a network interface. RDMA systems are not typically cache-coherent. 43–45, 126

service-level objective A target latency for an operation below which the user experiences little additional benefit. 76, 112, 126

SiP system in package 53

SoC system on chip 5, 8, 13–17, 19, 20, 22, 25, 36, 38, 42, 90, 114–116, 118, 121

swap Historically used to refer to the process of moving an entire process's memory image to disk, Linux uses "swapping" to refer to all paging. see 55

swap entry A Linux-specific value stored in evicted PTEs that contains information on where to locate an evicted page. 56, 57, 59

tail latency The worst case latency of some task. These are typically measured as percentiles (i.e., 99th percentile latency). 76, 103

task Linux kernel internal abstraction of a process. 50, 55

transcendent memory A layer in the Linux paging subsystem that stores pages in specialized memory that may not be disk-backed. 55–57, 59, 126

translation look-aside buffer A cache of virtual to physical address translations. 56, 57, 126

virtual memory area Contiguous region of virtual memory used by Linux to simplify memory management. 56–58, 127

warehouse-scale computer Generic term referring to tightly-integrated clusters of machines deployed in the datacenter. 1, 2, 5–9, 11–13, 22, 40, 73, 79, 86, 106, 107, 111, 116, 118, 119, 127

Acronyms

API application programming interface 8–11, 15, 18, 66, 86, 87, 92, 93, 95, 96, 106

cTask CPU task 88

DDC disaggregated datacenter 12

eTask Exclusive Task 92–95, 97, 98, 100, 101, 103

KaaS Kernel-as-a-Service 85, 86, 88–101, 103, 105–107, 110, 111, 113, 114, 119, 120, 122

kTask kernel task 86, 88–95, 97, 98, 100, 103, 107, 113

LRU least recently used 55, 57

NIC network interface card 41, 51–53, 60, 61

NUMA non-uniform memory access 43, 44

NVM non-volatile memory 44

PCSI portable cloud system interface 108–112, 114, 116, 122

PFA page fault accelerator 29–32, 47, 49–53, 55, 57–62, 64, 65, 113, 117, 118, 120–122

PGAS partitioned global address space 45

PTE page table entry 50–53, 56–58, 61

RDMA remote direct memory access 43–45

SLO service-level objective 76, 112

TLB translation look-aside buffer 56, 57

TMem transcendent memory 55–57, 59

VMA virtual memory area 56–58

WSC warehouse-scale computer 1, 2, 5–9, 11–13, 22, 40, 73, 79, 86, 106, 107, 111, 116, 118, 119

Bibliography

[1] Intel. Oct. 2021. URL: https://www.intel.com/content/www/us/en/products/sku/204087/intel-xeon-platinum-8380h-processor-38-5m-cache-2-90-ghz/specifications.html.

[2] Dihuni. Oct. 2021. URL: https://www.dihuni.com/product/nvidia-a100-80gb-900-21001-0020-000-gpu-pci-e-4-0/.

[3] *A Technical Overview of the Oracle Exadata Database Machine and Exadata Storage Server*. https://www.oracle.com/technetwork/database/exadata/exadata-technical-whitepaper-134575.pdf. Accessed: 2019-04-12. 2012.

[4] Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. "TensorFlow: A system for large-scale machine learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[5] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E.R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. "Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads". In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, May 2020, pp. 145–158. ISBN: 978-1-72814-661-4. DOI: 10.1109/ISCA45697.2020.00023. URL: https://ieeexplore.ieee.org/document/9138986/.

[6] Marcos K. Aguilera, Nadav Amit, Irina Calciu, Xavier Deguillard, Jayneel Gandhi, Stanko Novaković, Arun Ramanathan, Pratap Subrahmanyam, Lalith Suresh, Kiran Tati, Rajesh Venkatasubramanian, and Michael Wei. "Remote regions: a simple abstraction for remote memory". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 775–787. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/aguilera.

[7] William Allcock, Bennett Bernardoni, Colleen Bertoni, Neil Getty, Joseph Insley, Michael E. Papka, Silvio Rizzi, and Brian Toonen. "RAM as a Network Managed Resource". In: *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2018, pp. 99–106. DOI: 10.1109/IPDPSW.2018.00024.

[8] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. "The Eden system: A technical review". In: *IEEE Transactions on Software Engineering* (1985), pp. 43–59.

[9] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K. Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. "Can Far Memory Improve Job Throughput?" In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387522. URL: https://doi.org/10.1145/3342195.3387522.

[10] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanović, and Borivoje Nikolić. "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs". In: *IEEE Micro* 40.4 (2020), pp. 10–21. ISSN: 1937-4143. DOI: 10.1109/MM.2020.2996616.

[11] Alon Amid, Albert Ou, Krste Asanović, Yakun Sophia Shao, and Borivoje Nikolić. "Vertically Integrated Computing Labs Using Open-Source Hardware Generators and Cloud-Hosted FPGAs". In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. May 2021, pp. 1–5. DOI: 10.1109/ISCAS51556.2021.9401515.

[12] T. E. Anderson, D. E. Culler, and D. A. Patterson. "The Berkeley Networks of Workstations (NOW) Project". In: *Digest of Papers. COMPCON'95. Technologies for the Information Superhighway*. Mar. 1995, pp. 322–326. DOI: 10.1109/CMPCON.1995.512403.

[13] Jeremy Andrews and Ingo Molnar. *Interview: Ingo Molnar*. Dec. 2002. URL: https://web.archive.org/web/20070329030217/http://kerneltrap.org/node/517.

[14] *Ariane SDK*. URL: https://github.com/pulp-platform/ariane-sdk.

[15] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. "A view of cloud computing". In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[16] *Artifact Review and Badging*. Version 1.1. Association of Computer Machinery. Aug. 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current.

[17] Krste Asanović. "FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers". In: *FAST 2014*. 2014.

[18] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David A. Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[19] Krste Asanović and David A. Patterson. *Instruction Sets Should Be Free: The Case For RISC-V*. Tech. rep. UCB/EECS-2014-146. EECS Department, University of California, Berkeley, Aug. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html.

[20] *AutoYaST*. SUSE. URL: https://doc.opensuse.org/projects/autoyast/.

[21] Jonathan Bachrach. *Aspire RISC-V Rocket Accelerators – Lecture 02*. https://inst.eecs.berkeley.edu/~cs250/fa14/lectures/lec02.pdf. University of California, Berkeley, Sept. 2014.

[22] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avizienis, John Wawrzynek, and Krste Asanović. "Chisel: Constructing hardware in a Scala embedded language". In: *DAC Design Automation Conference 2012*. June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[23] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. "PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 499–514. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/bai.

[24] Henry C. Baker and Carl Hewitt. "The Incremental Garbage Collection of Processes". In: *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. New York, NY, USA: Association for Computing Machinery, 1977, pp. 55–59. ISBN: 9781450378741. DOI: 10.1145/800228.806932. URL: https://doi.org/10.1145/800228.806932.

[25] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, Dave Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony Rowstron. "Pelican: A Building Block for Exascale Cold Data Storage". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 351–365. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/balakrishnan.

[26] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables DevOps: Migration to a cloud-native architecture". In: *IEEE Software* 33.3 (2016), pp. 42–52.

[27] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, Matthew Matl, and David Wentzlaff. "OpenPiton: An Open Source Manycore Research Framework". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. Atlanta, Georgia, USA: ACM, 2016, pp. 217–232. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872414. URL: http://doi.acm.org/10.1145/2872362.2872414.

[28] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. "Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. Bordeaux, France: Association for Computing Machinery, 2015. ISBN: 9781450332385. DOI: 10.1145/2741948.2741962. URL: https://doi.org/10.1145/2741948.2741962.

[29] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization". In: *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles.* ACM, 2003, pp. 164–177. ISBN: 1-58113-757-5. DOI: 10.1145/945445.945462. URL: http://doi.acm.org/10.1145/945445.945462.

[30] Jeff Barr. "AQUA (Advanced Query Accelerator) – A Speed Boost for Your Amazon Redshift Queries". In: *AWS News Blog* (Apr. 2021). URL: https://aws.amazon.com/blogs/aws/new-aqua-advanced-query-accelerator-for-amazon-redshift/.

[31] Jeff Barr. "New – EC2 Instances (A1) Powered by Arm-Based AWS Graviton Processors". In: *AWS News Blog* (2018). URL: https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-processors/.

[32] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. "Attack of the Killer Microseconds". In: *Communications of the ACM* 60.4 (Mar. 2017), pp. 48–54. ISSN: 0001-0782. DOI: 10.1145/3015146. URL: https://doi.org/10.1145/3015146.

[33] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. "Web Search for a Planet: The Google Cluster Architecture". In: *IEEE Micro* 23.2 (Mar. 2003), pp. 22–28. ISSN: 0272-1732. DOI: 10.1109/MM.2003.1196112. URL: https://doi.org/10.1109/MM.2003.1196112.

[34] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. "The datacenter as a computer: Designing warehouse-scale machines". In: *Synthesis Lectures on Computer Architecture* 13.3 (2018), pp. i–189.

[35] Jim Basney and Miron Livny. "Managing network resources in Condor". In: *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. 2000, pp. 298–299. DOI: 10.1109/HPDC.2000.868666.

[36] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: Association for Computing Machinery, 2009, pp. 29–44. ISBN: 9781605587523. DOI: 10.1145/1629575.1629579. URL: https://doi.org/10.1145/1629575.1629579.

[37] Pete Beckman, Kamil Iskra, Kazutomo Yoshii, Susan Coghlan, and Aroon Nataraj. "Benchmarking the Effects of Operating System Interference on Extreme-scale Parallel Machines". In: *Cluster Computing* 11.1 (Mar. 2008), pp. 3–16. ISSN: 1386-7857. DOI: 10.1007/s10586-007-0047-2.

[38] Abhinav Bhatele, Kathryn Mohror, Steven H. Langer, and Katherine E. Isaacs. "There goes the neighborhood: performance degradation due to nearby jobs". In: ACM Press, 2013, pp. 1–12. ISBN: 978-1-4503-2378-9. DOI: 10.1145/2503210.2503247. URL: http://dl.acm.org/citation.cfm?doid=2503210.2503247.

[39] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. "Shredder: GPU-accelerated incremental storage and computation." In: *FAST*. Vol. 14. 2012, p. 14.

[40] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. "The End of Slow Networks: It's Time for a Redesign". In: *Proc. VLDB Endow.* 9.7 (Mar. 2016), pp. 528–539. ISSN: 2150-8097. DOI: 10.14778/2904483.2904485.

[41] D. Bonachea and P. Hargrove. *GASNet Specification, v1.8.1*. Tech. rep. LBNL-2001064. Lawrence Berkeley National Laboratory, 2017.

[42] Ali Borji. *Negative Results in Computer Vision: A Perspective*. 2017. DOI: 10.48550/ARXIV.1705.04402. URL: https://arxiv.org/abs/1705.04402.

[43] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Second Edition*. Ed. by Andy Oram. 3rd ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2005. ISBN: 978-0596005658.

[44] Eric A. Brewer. "Kubernetes and the path to cloud native". In: *Proceedings of the sixth ACM symposium on cloud computing*. 2015, pp. 167–167.

[45] Benjamin Brock, Aydın Buluç, Timothy Mattson, Scott McMillan, and José Moreira. *The GraphBLAS C API Specification*. 2021. URL: https://graphblas.org/docs/GraphBLAS_API_C_v2.0.0.pdf.

[46] *Buildroot*. Buildroot Association. URL: https://buildroot.org/.

[47] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. "Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade". In: *Queue* 14.1 (2016), pp. 70–93.

[48] Rajkumar Buyya, Toni Cortes, and Hai Jin. "Single system image". In: *The International Journal of High Performance Computing Applications* 15.2 (2001), pp. 124–135.

[49] *Cadence Palladium XP II Verification Computing Platform*. Tech. rep. Cadence Design Systems, Inc., 2013.

[50] Qingchao Cai, Wentian Guo, Hao Zhang, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, Yong Meng Teo, and Sheng Wang. "Efficient Distributed Memory Management with RDMA and Caching". In: *Proc. VLDB Endow.* 11.11 (July 2018), pp. 1604–1617. ISSN: 2150-8097. DOI: 10.14778/3236187.3236209. URL: https://doi.org/10.14778/3236187.3236209.

[51] Irina Calciu, M. Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. "Rethinking Software Runtimes for Disaggregated Memory". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 79–92. ISBN: 9781450383172. DOI: 10.1145/3445814.3446713. URL: https://doi.org/10.1145/3445814.3446713.

[52] Blake Caldwell, Sepideh Goodarzy, Sangtae Ha, Richard Han, Eric Keller, Eric Rozner, and Youngbin Im. "FluidMem: Full, Flexible, and Fast Memory Disaggregation for the Cloud". In: *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*. 2020, pp. 665–677. DOI: 10.1109/ICDCS47774.2020.00090.

[53] Amanda Carbonari and Ivan Beschasnikh. "Tolerating Faults in Disaggregated Datacenters". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. HotNets-XVI. Palo Alto, CA, USA: ACM, 2017, pp. 164–170. ISBN: 978-1-4503-5569-8. DOI: 10.1145/3152434.3152447. URL: http://doi.acm.org/10.1145/3152434.3152447.

[54] William W. Carlson, Jesse M. Draper, David E. Culler, Kathy Yelick, Eugene Brooks, and Karen Warren. *Introduction to UPC and language specification*. Tech. rep. CCS-TR-99-157. IDA Center for Computing Sciences, 1999.

[55] João Carreira. "Disaggregation in the Cloud with uInstances and Cirrus". In: *SOSP 2017 Student Research Competition*. 2017.

[56] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. "The rise of serverless computing". In: *Communications of the ACM* 62.12 (2019), pp. 44–54.

[57] *Cache Coherent Interconnect for Accelerators*. https://www.ccixconsortium.com. Accessed: 2019-04-12. 2017.

[58] Christopher Celio, David A. Patterson, and Krste Asanović. *The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor*. Tech. rep. UCB/EECS-2015-167.

EECS Department, University of California, Berkeley, June 2015. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-167.html.

[59] Christopher Celio, Andrew Waterman, Dabbelt Palmer, and David Biancolin. *Speckle*. URL: https://github.com/ccelio/Speckle/tree/firesim-2017.

[60] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeff Kuehn, Chuck Koelbel, and Lauren Smith. "Introducing OpenSHMEM: SHMEM for the PGAS community". In: *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model - PGAS '10*. ACM Press, 2010, pp. 1–3. ISBN: 978-1-4503-0461-0. DOI: 10.1145/2020373.2020375. URL: http://dl.acm.org/citation.cfm?doid=2020373.2020375.

[61] Elaine Chen. "What is a good ratio between developers and software quality assurance people?" In: (2019). URL: https://orbit-kb.mit.edu/hc/en-us/articles/206445976-What-is-a-good-ratio-between-developers-and-software-quality-assurance-people-.

[62] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/chen.

[63] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. "FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1077–1091. ISBN: 9781450371025. URL: https://doi.org/10.1145/3373376.3378515.

[64] David Cheriton. "The V distributed system". In: *Communications of the ACM* 31.3 (1988), pp. 314–333.

[65] Eric Chung, Jeremy Fowers, Kalin Ovtcharov, Adrian Caulfield, Todd Massengill, Ming Liu, Mahdi Ghandi, Daniel Lo, Steve Reinhardt, Shlomi Alkalay, Hari Angepat, Derek Chiou, Alessandro Forin, Doug Burger, Lisa Woods, Gabriel Weisz, Michael Haselman, and Dan Zhang. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". In: *IEEE Micro* 38 (Mar. 2018), pp. 8–20. URL: https://www.microsoft.com/en-us/research/publication/serving-dnns-real-time-datacenter-scale-project-brainwave/.

[66] Walfredo Cirne and Francine Berman.
"A Comprehensive Model of the Supercomputer Workload". In: *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*. WWC '01. USA: IEEE Computer Society, 2001, pp. 140–148. ISBN: 0780373154.

[67] *CoHMM Proxy Application: Multi-scale proxy app to test and evaluate runtime systems*. Extreme Materials at Extreme Scale.
URL: http://www.exmatex.org/cohmm.html.

[68] Christian Collberg and Todd A. Proebsting.
"Repeatability in Computer Systems Research".
In: *Commun. ACM* 59.3 (Feb. 2016), pp. 62–69. ISSN: 0001-0782.
DOI: 10.1145/2812803. URL: https://doi.org/10.1145/2812803.

[69] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. "Tessellation: Refactoring the OS around Explicit Resource Containers with Continuous Adaptation".
In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. Austin, Texas: Association for Computing Machinery, 2013. ISBN: 9781450320719.
DOI: 10.1145/2463209.2488827.
URL: https://doi.org/10.1145/2463209.2488827.

[70] Jonathan Corbet. "Preparing for user-space checkpoint/restore".
In: *Linux Weekly News* (2012). URL: https://lwn.net/Articles/478111/.

[71] *cuBLAS Library User Guide*. DU-06702-001_v11.5. Nvidia. Oct. 2021.
URL: https://docs.nvidia.com/cuda/pdf/CUBLAS_Library.pdf.

[72] *cuDNN Library Developer Guide*. PG-06702-001_v8.2.4. Nvidia. Aug. 2021.
URL: https://docs.nvidia.com/deeplearning/cudnn/pdf/cuDNN-Developer-Guide.pdf.

[73] *CUTLASS*. Version 2.7. Nvidia. URL: https://github.com/NVIDIA/cutlass.

[74] Partha Dasgupta, Richard J LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. "The Clouds distributed operating system".
In: *Computer* 24.11 (1991), pp. 34–44.

[75] Jeffrey Dean and Luiz André Barroso. "The tail at scale".
In: *Communications of the ACM* 56.2 (2013), pp. 74–80.

[76] *Debian – The Universal Operating System*. Software in the Public Interest, Inc.
URL: https://www.debian.org/.

[77] *Dell EMC PowerScale Hybrid Family*. Tech. rep. Dell Inc, 2021. URL: https://www.dell.com/en-us/dt/storage/powerscale/powerscale-hybrid-nas-storage.htm#scroll=off&pdf-overlay=//www.delltechnologies.com/asset/en-us/products/storage/technical-support/h16071-ss-powerscale-hybrid-nodes.pdf.

[78] *Deploy ML models to field-programmable gate arrays (FPGAs) with Azure Machine Learning*. Microsoft, Oct. 2021. URL: https://docs.microsoft.com/en-us/azure/machine-learning/how-to-deploy-fpga-web-service.

[79] *Deploying HPC Cluster with Mellanox InfiniBand Interconnect Solutions*. 2017. URL: http://www.mellanox.com/related-docs/solutions/deploying-hpc-cluster-with-mellanox-infiniband-interconnect-solutions-archive.pdf.

[80] M. Dorr, N. Barton, J. Keasler, and F. Li. *CoEVP: A Co-design Embedded ViscoPlasticity Scale-bridging Proxy App for ExMatEx*. Tech. rep. LLNL-SM-655180. Lawrence Livermore National Lab, June 2014. URL: https://github.com/exmatex/CoEVP/blob/master/CoEVP.pdf.

[81] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. "No Compromises: Distributed Transactions with Consistency, Availability, and Performance". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015, pp. 54–70. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815425. URL: http://doi.acm.org/10.1145/2815400.2815425.

[82] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. "rCUDA: Reducing the number of GPU-based accelerators in high performance clusters". In: *2010 International Conference on High Performance Computing Simulation*. 2010, pp. 224–231. DOI: 10.1109/HPCS.2010.5547126.

[83] Jason Duell. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*. Tech. rep. LBNL-54941. Lawrence Berkeley National Lab, 2002. URL: https://crd.lbl.gov/assets/pubs_presos/CDS/FTG/Papers/2002/blcr.pdf.

[84] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. "Exokernel: An operating system architecture for application-level resource management". In: *ACM SIGOPS Operating Systems Review* 29.5 (1995), pp. 251–266.

[85] Facebook. *Disaggregated Rack*. https://web.archive.org/web/20160421092139/http://www.opencompute.org/wp/wp-content/uploads/2013/01/OCP_Summit_IV_Disaggregation_Jason_Taylor.pdf. Accessed: 2019-04-12. 2013.

[86] Daniele Fanelli.
"Negative results are disappearing from most disciplines and countries".
In: *Scientometrics* 90 (2012), pp. 891–904.
DOI: https://doi.org/10.1007/s11192-011-0494-7.

[87] *Fedora*. Red Hat Inc. URL: https://start.fedoraproject.org/.

[88] Xinyang Feng, Jianjing Shen, and Ying Fan.
"REST: An alternative to RPC for Web services architecture".
In: *2009 First International Conference on Future Information Networks*.
IEEE. 2009, pp. 7–10.

[89] Kurt B. Ferreira, Patrick G. Bridges, Ron Brightwell, and Kevin T. Pedretti.
"The Impact of System Design Parameters on Application Noise Sensitivity".
In: *Cluster Computing* 16.1 (Mar. 2013), pp. 117–129. ISSN: 1386-7857.
DOI: 10.1007/s10586-011-0178-3.
URL: https://doi.org/10.1007/s10586-011-0178-3.

[90] *FireMarshal*. Nathan Pemberton.
URL: https://github.com/firesim/FireMarshal.

[91] *FlashBlade Unified Fast File and Object Storage*. Tech. rep. PureStorage, 2021.
URL: https://www.purestorage.com/docs.html?item=/type/pdf/subtype/doc/path/content/dam/pdf/en/solution-briefs/sb-pure-flashblade-uffo.pdf.

[92] B. Fleisch and G. Popek.
"Mirage: A Coherent Distributed Shared Memory Design".
In: *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*.
SOSP '89. New York, NY, USA: Association for Computing Machinery, 1989,
pp. 211–223. ISBN: 0897913388. DOI: 10.1145/74850.74871.
URL: https://doi.org/10.1145/74850.74871.

[93] Alessandro Forin, Joseph Barrera, and Richard Sanzi.
"The Shared Memory Server". In: *Proceedings of the 1989 Winter Unix Conference*.
1989, pp. 229–243.

[94] *freedom-u-sdk*. Sifive, Inc.
URL: https://github.com/sifive/freedom-u-sdk/tree/2020.09.

[95] Paolo Gai, Luca Abeni, Massimiliano Giorgi, and Giorgio Buttazzo.
"A new kernel approach for modular real-time systems development".
In: *Proceedings 13th Euromicro Conference on Real-Time Systems*. IEEE. 2001,
pp. 199–206.

[96] Shay Gal-On and Markus Levy.
Exploring CoreMark™ – A Benchmark Maximizing Simplicity and Efficacy.
EEMBC.
URL: https://www.eembc.org/techlit/articles/coremark-whitepaper.pdf.

[97] Peter X. Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. "Network Requirements for Resource Disaggregation". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 249–264. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao.

[98] Gen-Z Consortium. *Gen-Z Overview*. Tech. rep. Gen-Z Consortium, 2016.

[99] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. "Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration". In: *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 2021, pp. 769–774. DOI: 10.1109/DAC18074.2021.9586216.

[100] Mark Giampapa, Thomas Gooding, Todd Inglett, and Robert W. Wisniewski. "Experiences with a Lightweight Supercomputer Kernel: Lessons Learned from Blue Gene's CNK". In: *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '10. USA: IEEE Computer Society, 2010, pp. 1–10. ISBN: 9781424475599. DOI: 10.1109/SC.2010.22. URL: https://doi.org/10.1109/SC.2010.22.

[101] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. "Replicating and Mitigating Spectre Attacks on a Open Source RISC-V Microarchitecture". In: *Second Workshop on Computer Architecture Research with RISC-V*. CARRV '19. Phoenix, Arizona, USA, June 2019.

[102] Cary Gray and David Cheriton. "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency". In: *ACM SIGOPS Operating Systems Review* 23.5 (1989), pp. 202–210.

[103] Haifeng Gu, Mingsong Chen, Yanzhao Wang, and Fei Xie. "SpectreCheck: An Approach to Detecting Speculative Execution Side Channels in Data Cache". In: *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*. 2020, pp. 1–8. DOI: 10.1109/ICESS49830.2020.9301601.

[104] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. "Efficient Memory Disaggregation with Infiniswap". In: *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 649–667. ISBN: 978-1-931971-37-9. URL: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[105] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. “Serving DNNs like Clockwork: Performance Predictability from the Bottom Up”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 443–462. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/gujarati.

[106] Jing Guo, Zihao Chang, Sa Wang, Haiyang Ding, Yihui Feng, Liang Mao, and Yungang Bao. “Who Limits the Resource Efficiency of My Datacenter: An Analysis of Alibaba Datacenter Traces”. In: *Proceedings of the International Symposium on Quality of Service*. IWQoS ’19. Phoenix, Arizona: Association for Computing Machinery, 2019. ISBN: 9781450367783. DOI: 10.1145/3326285.3329074. URL: https://doi.org/10.1145/3326285.3329074.

[107] Nadav Har’El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. “Efficient and Scalable Paravirtual I/O System”. In: *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. San Jose, CA: USENIX, 2013, pp. 231–242. ISBN: 978-1-931971-01-0. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/har%7B%5Ctextquoteright%7Del.

[108] Rober Haskin, Yoni Malachi, and Gregory Chan. “Recovery management in QuickSilver”. In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 82–108.

[109] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. “Serverless Computing: One Step Forward, Two Steps Back”. In: *CIDR*. 2019.

[110] A. J. Herbert. “The Cambridge Distributed Computing System”. In: *Proceedings on Local Area Networks: An Advanced Course*. Berlin, Heidelberg: Springer-Verlag, 1983, pp. 282–312. ISBN: 3540151915.

[111] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: A correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[112] Torsten Hoefler, Torsten Mehlan, Andrew Lumsdaine, and Wolfgang Rehm. “Netgauge: A Network Performance Measurement Framework”. In: *Proceedings of High Performance Computing and Communications, HPCC’07*. Vol. 4782. Houston, USA: Springer, Sept. 2007, pp. 659–671. ISBN: 978-3-540-75443-5. URL: http://www.unixer.de/~htor/publications/.

[113] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. "Scale and performance in a distributed file system". In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 51–81.

[114] HP Labs. *The Machine*. https://www.labs.hpe.com/the-machine. Accessed: 2019-04-12. 2017.

[115] Joel Hruska. "As Chip Design Costs Skyrocket, 3nm Process Node Is in Jeopardy". In: *ExtremeTech* (June 2018). URL: https://www.extremetech.com/computing/272096-3nm-process-node.

[116] Jeremy Hsu. "RISC-V Star Rises Among Chip Developers Worldwide: The upstart RISC-V chip architecture has found international traction with its customizable open-source design and lack of licensing fees". In: *IEEE Spectrum* (Apr. 2021). URL: https://spectrum.ieee.org/riscv-rises-among-chip-developers-worldwide.

[117] Qijing Huang, Christopher Yarp, Sagar Karandikar, Nathan Pemberton, Benjamin Brock, Liang Ma, Guohao Dai, Robert Quitt, Krste Asanovic, and John Wawrzynek. "Centrifuge: Evaluating full-system HLS-generated heterogeneous-accelerator SoCs using FPGA-Acceleration". In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2019, p. 8.

[118] Huawei. *High Throughput Computing Data Center Architecture - Thinking of Data Center 3.0*. https://www.huawei.com/ilink/en/download/HW_349607. Accessed: 2019-04-12. 2014.

[119] *Inferentia*. Amazon. 2022. URL: https://aws.amazon.com/machine-learning/inferentia/.

[120] Intel. *Intel Rack Scale Design*. https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html. Accessed: 2019-04-12. 2017.

[121] Peter Ivie and Douglas Thain. "Reproducibility in Scientific Computing". In: *ACM Comput. Surv.* 51.3 (July 2018). ISSN: 0360-0300. DOI: 10.1145/3186266. URL: https://doi.org/10.1145/3186266.

[122] *Jacobi CUDA Graphs*. NVIDIA. URL: https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/jacobiCudaGraphs.

[123] Myeongjae Jeon, Shivaram Venkataraman, Amar Phanishayee, Junjie Qian, Wencong Xiao, and Fan Yang. "Analysis of Large-Scale Multi-Tenant GPU Clusters for DNN Training Workloads". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 947–960. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/jeon.

[124] Morris A. Jette. “Performance Characteristics of Gang Scheduling in Multiprogrammed Environments”. In: *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing.* SC ’97. San Jose, CA: Association for Computing Machinery, 1997, pp. 1–12. ISBN: 0897919858. DOI: 10.1145/509593.509647. URL: https://doi.org/10.1145/509593.509647.

[125] M. Jones. *Virtio: An I/O virtualization framework for Linux.* Tech. rep. IBM, Jan. 2010. URL: https://developer.ibm.com/articles/l-virtio/.

[126] Terry Jones, Shawn Dawson, Rob Neely, William Tuel, Larry Brenner, Jeffrey Fier, Robert Blackmore, Patrick Caffrey, Brian Maskell, Paul Tomlinson, and Mark Roberts. “Improving the Scalability of Parallel Jobs by Adding Parallel Awareness to the Operating System”. In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing.* SC ’03. Phoenix, AZ, USA: Association for Computing Machinery, 2003, p. 10. ISBN: 1581136951. DOI: 10.1145/1048935.1050161. URL: https://doi.org/10.1145/1048935.1050161.

[127] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. “In-Datacenter Performance Analysis of a Tensor Processing Unit”. In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 1–12. ISSN: 0163-5964. DOI: 10.1145/3140659.3080246. URL: https://doi.org/10.1145/3140659.3080246.

[128] *Juice Labs.* 2021. URL: https://www.juicelabs.co/.

[129] Anuj Kalia, Michael Kaminsky, and David G. Andersen. “FaSST: Fast, Scalable and Simple Distributed Transactions with Two-Sided (RDMA) Datagram RPCs”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16).* Savannah, GA: USENIX Association, 2016, pp. 185–201.

isbn: 978-1-931971-33-1.
url: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kalia.

[130] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. "Profiling a warehouse-scale computer". In: ACM Press, 2015, pp. 158–169. isbn: 978-1-4503-3402-0. doi: 10.1145/2749469.2750392. url: http://dl.acm.org/citation.cfm?doid=2749469.2750392.

[131] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. "HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture.* ACM, 2017, pp. 521–534. url: http://pages.cs.wisc.edu/~sudarsun/ISCA17_HeteroOS_Kannan.pdf.

[132] Antti Kantee. *On Rump Kernels and the Rumprun Unikernel.* Tech. rep. The Linux Foundation, 2015. url: https://xenproject.org/2015/08/06/on-rump-kernels-and-the-rumprun-unikernel/.

[133] Sagar Karandikar, Chris Leary, Chris Kennelly, Jerry Zhao, Dinesh Parimi, Borivoje Nikolic, Krste Asanovic, and Parthasarathy Ranganathan. "A Hardware Accelerator for Protocol Buffers". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture.* MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 462–478. isbn: 9781450385572. doi: 10.1145/3466752.3480051. url: https://doi.org/10.1145/3466752.3480051.

[134] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Bora Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. "FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud". In: *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA).* June 2018, pp. 29–42. doi: 10.1109/ISCA.2018.00014.

[135] K. Katrinis, D. Syrivelis, D. Pnevmatikatos, G. Zervas, D. Theodoropoulos, I. Koutsopoulos, K. Hasharoni, D. Raho, C. Pinto, F. Espina, S. Lopez-Buedo, Q. Chen, M. Nemirovsky, D. Roca, H. Klos, and T. Berends. "Rack-scale disaggregated cloud data centers: The dReDBox project vision". In: *2016 Design, Automation Test in Europe Conference Exhibition (DATE).* Mar. 2016, pp. 690–695.

[136] Kimberly Keeton. "Memory-Driven Computing". In: *15th USENIX Conference on File and Storage Technologies (FAST '17).* Santa Clara, CA: USENIX Association, Feb. 2017.

[137] Kimberly Keeton, Sharad Singhal, and Michael Raymond. "The OpenFAM API: A Programming Model for Disaggregated Persistent Memory". In: *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Extreme Heterogeneity.* Ed. by Swaroop Pophale, Neena Imam, Ferrol Aderholdt, and Manjunath Gorentla Venkata. Springer International Publishing, 2019, pp. 70–89. ISBN: 978-3-030-04918-8.

[138] Kimberly Keeton, Sharad Singhal, Haris Volos, Yupu Zhang, Ramesh Chandra Chaurasiya, Clarete Riana Crasta, Sherin T George, Nagaraju K N, Mashood Abdulla K, Kavitha Natarajan, Porno Shome, and Sanish Suresh. "MODC: Resilience for disaggregated memory architectures using task-based programming". In: (Apr. 2021). URL: https://wuklab.github.io/words/words21-keeton.pdf.

[139] Suzanne M. Kelly and Ron Brightwell. "Software architecture of the light weight kernel, Catamount". In: *Proceedings of the 2005 Cray User Group Annual Technical Conference.* Citeseer, 2005, pp. 16–19. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.5559&rep=rep1&type=pdf.

[140] Jaewook Kim, Tae Joon Jun, Daeyoun Kang, Dohyeun Kim, and Daeyoung Kim. "GPU Enabled Serverless Computing Framework". In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP).* Mar. 2018, pp. 533–540. DOI: 10.1109/PDP2018.2018.00090.

[141] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18).* Carlsbad, CA: USENIX Association, 2018, pp. 427–444. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[142] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution". In: *2019 IEEE Symposium on Security and Privacy (SP).* 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[143] Dario Korolija, Dimitrios Koutsoukos, Kimberly Keeton, Konstantin Taranov, Dejan Milojičić, and Gustavo Alonso. "Farview: Disaggregated Memory with Operator Off-loading for Database Engines". In: *Conference on Innovative Data Systems Research (CIDR).* 2022.

[144] Shriram Krishnamurthi and Jan Vitek. "The Real Software Crisis: Repeatability as a Core Value". In: *Commun. ACM* 58.3 (Feb. 2015), pp. 34–36. ISSN: 0001-0782. DOI: 10.1145/2658987. URL: https://doi.org/10.1145/2658987.

[145] John Lange, Kevin Pedretti, Trammell Hudson, Peter Dinda, Zheng Cui, Lei Xia, Patrick Bridges, Andy Gocke, Steven Jaconette, Mike Levenhagen, and Ron Brightwell. "Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing". In: *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*. Apr. 2010, pp. 1–12. DOI: 10.1109/IPDPS.2010.5470482.

[146] James Laudon and Daniel Lenoski. "The SGI Origin: A ccNUMA Highly Scalable Server". In: *Proceedings of the 24th Annual International Symposium on Computer Architecture*. ISCA '97. ACM, 1997, pp. 241–251. ISBN: 978-0-89791-901-2. DOI: 10.1145/264107.264206. URL: http://doi.acm.org/10.1145/264107.264206.

[147] Nikita Lazarev, Shaojie Xiang, Neil Adit, Zhiru Zhang, and Christina Delimitrou. "Dagger: Efficient and Fast RPCs in Cloud Microservices with near-Memory Reconfigurable NICs". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS 2021. Virtual, USA: Association for Computing Machinery, 2021, pp. 36–51. ISBN: 9781450383172. DOI: 10.1145/3445814.3446696. URL: https://doi.org/10.1145/3445814.3446696.

[148] Collin Lee and John Ousterhout. "Granular Computing". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 149–154.

[149] Cynthia Bailey Lee and Allan Snavely. "On the User - Scheduler Dialogue: Studies of User-Provided Runtime Estimates and Utility Functions". In: *IJHPCA* 20.4 (2006), pp. 495–506. DOI: 10.1177/1094342006068414. URL: http://dx.doi.org/10.1177/1094342006068414.

[150] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. "Keystone: An Open Framework for Architecting Trusted Execution Environments". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. 2020.

[151] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W. Lee. "A Fully Associative, Tagless DRAM Cache". In: *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*. ISCA '15. New York, NY, USA: ACM, 2015, pp. 211–222. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750383. URL: http://doi.acm.org/10.1145/2749469.2750383.

[152] Jacob Leverich and Christos Kozyrakis.
"Reconciling high server utilization and sub-millisecond quality-of-service". In:
ACM Press, 2014, pp. 1–14. ISBN: 978-1-4503-2704-6.
DOI: 10.1145/2592798.2592821.
URL: http://dl.acm.org/citation.cfm?doid=2592798.2592821.

[153] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini.
"First-generation Memory Disaggregation for Cloud Platforms".
In: *arXiv:2203.00241 [cs]* (Mar. 2022). arXiv: 2203.00241.
URL: http://arxiv.org/abs/2203.00241.

[154] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble.
"Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency".
In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14.
ACM, 2014, 9:1–9:14. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670988.
URL: http://doi.acm.org/10.1145/2670979.2670988.

[155] Sean Lie. "Wafer-Scale Deep Learning". In: *HotChips 2019*. 2019.

[156] Anthony Liguori. *C5 Instances and the Evolution of Amazon EC2 Virtualization*.
https://www.youtube.com/watch?v=LabltEXk0VQ. Accessed: 2019-04-12. 2017.

[157] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch.
"Disaggregated Memory for Expansion and Sharing in Blade Servers". In:
Proceedings of the 36th Annual International Symposium on Computer Architecture.
ISCA '09. Austin, TX, USA: ACM, 2009, pp. 267–278. ISBN: 978-1-60558-526-0.
DOI: 10.1145/1555754.1555789.
URL: http://doi.acm.org/10.1145/1555754.1555789.

[158] Kevin Lim, Yoshio Turner, Jose Renato Santos, Alvin AuYoung, Jichuan Chang, Parthasarathy Ranganathan, and Thomas F. Wenisch.
"System-level implications of disaggregated memory".
In: *IEEE International Symposium on High-Performance Comp Architecture*.
IEEE, 2012, pp. 1–12.
URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6168955.

[159] *What is Linux Memory Policy?* Linux Kernel Organization, Inc. 2017. URL:
https://www.kernel.org/doc/Documentation/vm/numa_memory_policy.txt.

[160] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li.
"Octopus: an RDMA-enabled Distributed Persistent Memory File System".
In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*.
Santa Clara, CA: USENIX Association, July 2017, pp. 773–785.

ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lu.

[161] Chris Lumens. *Kickstart*. URL: https://pykickstart.readthedocs.io/en/latest/index.html.

[162] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. "Rammer: Enabling Holistic Deep Learning Compiler Optimizations with rTasks". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 881–897. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ma.

[163] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. "AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: Association for Computing Machinery, 2020, pp. 757–773. ISBN: 9781450371025. URL: https://doi.org/10.1145/3373376.3378511.

[164] Anil Madhavapeddy and David J Scott. "Unikernels: the rise of the virtual library operating system". In: *Communications of the ACM* 57.1 (2014), pp. 61–69.

[165] Dan Magenheimer. "Transcendent memory in a nutshell". In: *Linux Weekly News* (2011). URL: https://lwn.net/Articles/454795/.

[166] Albert Magyar, David Biancolin, John Koenig, Sanjit A. Seshia, Jonathan Bachrach, and Krste Asanović. "Golden Gate: Bridging The Resource-Efficiency Gap Between ASICs and FPGA Prototypes". In: *In Proceedings of the International Conference on Computer-Aided Design (ICCAD)*. Nov. 2019, pp. 1–8.

[167] Neethu Bal Mallya, Cecilia Gonzalez-Alvarez, and Trevor E Carlson. "Flexible Timing Simulation of RISC-V Processors with Sniper". In: *Second Workshop on Computer Architecture Research with RISC-V*. CARRV '18. Los Angeles, California, USA.

[168] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. "Agile SoC Development with Open ESP". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450380263. DOI: 10.1145/3400302.3415753. URL: https://doi.org/10.1145/3400302.3415753.

[169] Howard Mao.
"Designing New Memory Systems for Next-Generation Data Centers".
PhD thesis. University of California at Berkeley, 2020.

[170] Howard Mao, Randy H Katz, and Krste Asanović.
"Hardware acceleration for memory to memory copies".
MA thesis. EECS Department, University of California, 2017.

[171] Hasan Al Maruf and Mosharaf Chowdhury.
"Effectively Prefetching Remote Memory with Leap".
In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*.
USENIX Association, July 2020, pp. 843–857. ISBN: 978-1-939133-14-4.
URL: https://www.usenix.org/conference/atc20/presentation/al-maruf.

[172] Scott McFarling. *Combining Branch Predictors*. Tech. rep. TN-36.
Western Research Laboratory, 1993.

[173] Paul E McKenney and Jonathan Walpole. "What is RCU, Fundamentally?"
In: *Linux Weekly News* (2007). URL: https://lwn.net/Articles/262464/.

[174] *memcached: a distributed memory object caching system*. 2019.
URL: https://memcached.org/.

[175] Dirk Merkel.
"Docker: lightweight linux containers for consistent development and deployment".
In: *Linux journal* 2014.239 (2014), p. 2.

[176] George Michelogiannakis, Benjamin Klenk, Brandon Cook, Min Yee Teh,
Madeleine Glick, Larry Dennison, Keren Bergman, and John Shalf.
"A Case For Intra-rack Resource Disaggregation in HPC". In: *ACM Transactions on Architecture and Code Optimization* 19.2 (Mar. 2022), 29:1–29:26.
ISSN: 1544-3566. DOI: 10.1145/3514245.
URL: https://doi.org/10.1145/3514245.

[177] Dejan S. Milojicic, David L. Black, and Steven J. Sears.
"Operating System Support for Concurrent Remote Task Creation".
In: *Proceedings of the 9th International Symposium on Parallel Processing*.
IPPS '95. USA: IEEE Computer Society, 1995, p. 486. ISBN: 0818670746.

[178] *MIPS® Architecture For Programmers Volume II-A: The MIPS64® Instruction Set Reference Manual*. Version 6.06. MIPS. 2016.

[179] *Mission-critical infrastructure for the data-driven enterprise*. Tech. rep.
Hewlett-Packard Enterprise, 2020.

[180] *ModelSim*. Mentor, a Siemens Business.
URL: https://www.mentor.com/products/fv/modelsim/.

[181] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, et al. "Ray: A distributed framework for emerging AI applications". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 2018, pp. 561–577.

[182] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. "Amoeba: A distributed operating system for the 1990s". In: *Computer* 23.5 (1990), pp. 44–53.

[183] *Multi-Process Service*. Nvidia. Oct. 2021. URL: https://docs.nvidia.com/deploy/pdf/CUDA_Multi_Process_Service_Overview.pdf.

[184] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "Ciel: A universal execution engine for distributed data-flow computing". In: *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 2011, pp. 113–126.

[185] Theodore H. Myer and Ivan E. Sutherland. "On the Design of Display Processors". In: *Communications of the ACM* 11.6 (1968).

[186] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. "Latency-tolerant software distributed shared memory". In: *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. 2015, pp. 291–305. URL: https://www.usenix.org/node/190522.

[187] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. "Caching in the Sprite network file system". In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 134–154.

[188] *NetApp FAS9000 Modular Hybrid Flash System*. Tech. rep. NetApp Inc, 2019. URL: https://www.netapp.com/pdf.html?item=/media/8939-ds-3810.pdf.

[189] *NOOBS (New Out of Box Software)*. Version 3.0. Raspberry Pi. URL: https://github.com/raspberrypi/noobs/releases/tag/v3.0.

[190] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. "Scale-out NUMA". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 3–18. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541965.

[191] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. "A concurrent specification of POSIX file systems". In: *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2018.

[192] *Nuclio*. iguazio, 2021. URL: https://github.com/nuclio/nuclio.

[193] *NVIDIA Jetson Linux Developer Guide*. Version 32.4.3. NVIDIA Corporation. July 2020. URL: https://docs.nvidia.com/jetson/l4t/index.html.

[194] *NVIDIA Virtual Compute Server*. Tech. rep. Apr. 2021. URL: https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/nvidia-virtual-compute-server-solution-overview.pdf.

[195] *ONNX Runtime*. Microsoft Corporation. URL: https://microsoft.github.io/onnxruntime/.

[196] *OpenFaaS*. Version 0.21.1. OpenFaaS Ltd. URL: https://www.openfaas.com/.

[197] *OpenRISC 1000 Architecture Manual*. Version 1.3. OPENRISC.io. June 2019.

[198] *OpenSBI*. Western Digital Corproration. URL: https://github.com/riscv/opensbi.

[199] *OpenStack*. https://www.openstack.org/.

[200] John K Ousterhout. "Scheduling Techniques for Concurrent Systems." In: *ICDCS*. Vol. 82. 1982, pp. 22–30.

[201] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. "The Sprite network operating system". In: *Computer* 21.2 (1988), pp. 23–36.

[202] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. "Monotasks: Architecting for performance clarity in data analytics frameworks". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. 2017, pp. 184–200.

[203] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc., 2019. URL: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[204] Nathan Pemberton. "Enabling Efficient and Transparent Remote Memory Access in Disaggregated Datacenters". MA thesis. EECS Department, University of California, Berkeley, Dec. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-154.html.

[205] Nathan Pemberton and Alon Amid. "FireMarshal: Making HW/SW Co-Design Reproducible and Reliable". In: *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 2021.

[206] Nathan Pemberton and Johann Schleier-Smith. "The Serverless Data Center: Hardware Disaggregation Meets Serverless Computing". In: *First Workshop on Resource Disaggregation*. 2019.

[207] Nathan Pemberton, Johann Schleier-Smith, Joseph Gonzalez, and Joseph Hellerstein. "The RESTLess Cloud". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. Association for Computing Machinery, 2021.

[208] Larry Peterson, Scott Baker, Marc De Leenheer, Andy Bavier, Sapan Bhatia, Mike Wawrzoniak, Jude Nelson, and John Hartman. "XOS: An extensible cloud operating system". In: *Proceedings of the 2nd International Workshop on Software-Defined Ecosystems*. 2015, pp. 23–30.

[209] Fabrizio Petrini, Darren J. Kerbyson, and Scott Pakin. "The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q". In: *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. SC'03. ACM, 2003, p. 55. ISBN: 1-58113-695-1. DOI: 10.1145/1048935.1050204. URL: http://doi.acm.org/10.1145/1048935.1050204.

[210] Daniel Petrisko, Farzam Gilani, Mark Wyse, Tommy Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, Tavio Guarino, Ajay Joshi, Mark Oskin, and Michael Taylor. "BlackParrot: An Agile Open Source RISC-V Multicore for Accelerator SoCs". In: *IEEE Micro* 40.4 (2020), pp. 93–102. DOI: 10.1109/MM.2020.2996145.

[211] Fabio Pianese, Peter Bosch, Alessandro Duminuco, Nico Janssens, Thanos Stathopoulos, and Moritz Steiner. "Toward a cloud operating system". In: *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*. IEEE. 2010, pp. 335–342.

[212] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. "Plan 9 from Bell Labs". In: *Computing systems* 8.3 (1995), pp. 221–254.

[213] Lutz Prechelt. "Why We Need an Explicit Forum for Negative Results". In: *Journal of Universal Computer Science*. Vol. 3. 1997.

[214] *The RISC-V Proxy Kernel*. The Regents of the University of California, 2019. URL: https://github.com/riscv/riscv-pk.

[215] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 114–122. ISSN: 0001-0782. DOI: 10.1145/2996868. URL: https://doi.org/10.1145/2996868.

[216] *QEMU*. Version 5.0.0. Software Freedom Conservancy, Apr. 28, 2020. URL: https://github.com/qemu/qemu/tree/v5.0.0.

[217] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI '13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[218] Aditya Ramkumar. "Making the Most of Serverless Accelerators". MA thesis. EECS Department, University of California, Berkeley, May 2022.

[219] B. Randell. "A Note on Storage Fragmentation and Program Segmentation". In: *Commun. ACM* 12.7 (July 1969), 365–ff. ISSN: 0001-0782. DOI: 10.1145/363156.363158. URL: https://doi.org/10.1145/363156.363158.

[220] Parthasarathy Ranganathan, Daniel Stodolsky, Jeff Calow, Jeremy Dorfman, Marisabel Guevara, Clinton Wills Smullen IV, Aki Kuusela, Raghu Balasubramanian, Sandeep Bhatia, Prakash Chauhan, Anna Cheung, In Suk Chong, Niranjani Dasharathi, Jia Feng, Brian Fosco, Samuel Foss, Ben Gelb, Sara J. Gwin, Yoshiaki Hase, Da-ke He, C. Richard Ho, Roy W. Huffman Jr., Elisha Indupalli, Indira Jayaram, Poonacha Kongetira, Cho Mon Kyaw, Aaron Laursen, Yuan Li, Fong Lou, Kyle A. Lucke, JP Maaninen, Ramon Macias, Maire Mahony, David Alexander Munday, Srikanth Muroor, Narayana Penukonda, Eric Perkins-Argueta, Devin Persaud, Alex Ramirez, Ville-Mikko Rautio, Yolanda Ripley, Amir Salek, Sathish Sekar, Sergey N. Sokolov, Rob Springer, Don Stark, Mercedes Tan, Mark S. Wachsler, Andrew C. Walton, David A. Wickeraad, Alvin Wijaya, and Hon Kwan Wu. "Warehouse-scale video acceleration: co-design and deployment in the wild". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS 2021. Association for Computing Machinery, Apr. 2021, pp. 600–615.

ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446723.
URL: https://doi.org/10.1145/3445814.3446723.

[221] *Ray v1.6.0 Documentation*. URL: https://docs.ray.io/en/releases-1.6.0/using-ray-with-gpus.html#workers-not-releasing-gpu-resources.

[222] R. Recio, B. Metzler, P. Culley, J. Hilland, and D. Garcia. *A Remote Direct Memory Access Protocol Specification*. RFC 5040. RFC Editor, Oct. 2007.

[223] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. *MLPerf Inference Benchmark*. 2020. arXiv: 1911.02549 [cs.LG].

[224] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. SoCC '12. San Jose, California: ACM, 2012, 7:1–7:13. ISBN: 978-1-4503-1761-0. DOI: 10.1145/2391229.2391236. URL: http://doi.acm.org/10.1145/2391229.2391236.

[225] Burkhard Ringlein, Francois Abel, Alexander Ditter, Beat Weiss, Christoph Hagleitner, and Dietmar Fey. "System Architecture for Network-Attached FPGAs in the Cloud using Partial Reconfiguration". In: *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*. 2019, pp. 293–300. DOI: 10.1109/FPL.2019.00054.

[226] *riscvOVPsim*. Imperas Software, Ltd., Oct. 22, 2020. URL: https://github.com/riscv/riscv-ovpsim.

[227] Dennis M. Ritchie and Ken Thompson. "The UNIX Time-Sharing System". In: *Commun. ACM* 17.7 (July 1974), pp. 365–375. ISSN: 0001-0782. DOI: 10.1145/361011.361061. URL: https://doi.org/10.1145/361011.361061.

[228] *RoCE in the Data Center*. Oct. 2014. URL: http://www.mellanox.com/related-docs/whitepapers/roce_in_the_data_center.pdf.

[229] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. "Relay: A New IR for Machine Learning Frameworks". In: *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, pp. 58–68. ISBN: 9781450358347. DOI: 10.1145/3211346.3211348. URL: https://doi.org/10.1145/3211346.3211348.

[230] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. "INFaaS: Automated Model-less Inference Serving". In: *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, July 2021, pp. 397–411. ISBN: 978-1-939133-23-6. URL: https://www.usenix.org/conference/atc21/presentation/romero.

[231] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. "Dandelion: a compiler and runtime for heterogeneous systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, Nov. 2013, pp. 49–68. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522715. URL: https://dl.acm.org/doi/10.1145/2517349.2522715.

[232] Bernard Rous. *The ACM Task Force on Data, Software, and Reproducibility in Publication*. URL: https://www.acm.org/publications/task-force-on-data-software-and-reproducibility.

[233] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. "AIFM: High-Performance, Application-Integrated Far Memory". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*. USENIX Association, Nov. 2020, pp. 315–332. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ruan.

[234] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. "Log-structured Memory for DRAM-based Storage". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. Santa Clara, CA: USENIX, 2014, pp. 1–16. ISBN: ISBN 978-1-931971-08-9. URL: https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble.

[235] Majid Sabbagh, Yunsi Fei, and David Kaeli. "Secure Speculative Execution via RISC-V Open Hardware Design". In: *Fifth Workshop on Computer Architecture Research with RISC-V*. CARRV '21. Online, June 2021.

[236] Nosayba El-Sayed and Bianca Schroeder. "Reading between the lines of failure logs: Understanding how HPC systems fail". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575356.

[237] Eduardo Schettino. *pydoit*. URL: https://pydoit.org/.

[238] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton, and Joseph M. Hellerstein. *A FaaS File System for Serverless Computing*. 2020. DOI: 10.48550/ARXIV.2009.09845. URL: https://arxiv.org/abs/2009.09845.

[239] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. "What serverless computing is and should become: The next phase of cloud computing". In: *Communications of the ACM* 64.5 (2021), pp. 55–63.

[240] Frank Schmuck and Jim Wylie. "Experience with transactions in QuickSilver". In: *ACM SIGOPS Operating Systems Review*. Vol. 25. ACM. 1991, pp. 239–253.

[241] Malte Schwarzkopf. "Operating system support for warehouse-scale computing". PhD thesis. University of Cambridge, 2015.

[242] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. "New wine in old skins: The case for distributed operating systems in the data center". In: *Proceedings of the 4th Asia-Pacific Workshop on Systems*. 2013, pp. 1–7.

[243] André Seznec and Pierre Michaud. "A case for (partially) TAgged GEometric history length branch prediction". In: *Journal of Instruction-level Parallelism - JILP* 8 (Jan. 2006).

[244] *SGI UV 3000, UV 30: Big Brains for No-Limit Computing*. 2016. URL: https://www.sgi.com/pdfs/4555.pdf.

[245] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 205–218. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[246] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. "LegoOS: A Disseminated, Distributed OS for Hardware Resource Disaggregation". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, 2018, pp. 69–87. ISBN: 978-1-931971-47-8. URL: https://www.usenix.org/conference/osdi18/presentation/shan.

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

[248] Debendra Das Sharma and Siamak Tavallaei. *Compute Express Link™ 2.0 White Paper*. Tech. rep. CXL Consortium, 2022. URL: https://b373eaf2-67af-4a29-b28c-3aae9e644f30.filesusr.com/ugd/0c1418_14c5283e7f3e40f9b2955c7d0f60bebe.pdf.

[249] Edi Shmueli, George Almasi, Jose Brunheroto, Jose Castanos, Gabor Dozsa, Sameer Kumar, and Derek Lieber. "Evaluating the Effect of Replacing CNK with Linux on the Compute-Nodes of Blue Gene/l". In: *Proceedings of the 22nd Annual International Conference on Supercomputing*. ICS '08. Island of Kos, Greece: Association for Computing Machinery, 2008, pp. 165–174. ISBN: 9781605581583. DOI: 10.1145/1375527.1375554. URL: https://doi.org/10.1145/1375527.1375554.

[250] David Sidler, Zeke Wang, Monica Chiosa, Amit Kulkarni, and Gustavo Alonso. "StRoM: Smart Remote Memory". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387519. URL: https://doi.org/10.1145/3342195.3387519.

[251] SiFive. *SiFive TileLink Specification*. Aug. 2017. URL: https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf.

[252] Gagandeep Singh. *Gate-Level Simulation Methodology: Improving Gate-Level Simulation Performance*. Tech. rep. Cadence Design Systems, Inc., 2015.

[253] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. "DBOS: A DBMS-Oriented Operating System". In: *Proc. VLDB Endow.* 15.1 (Sept. 2021), pp. 21–30. ISSN: 2150-8097. DOI: 10.14778/3485450.3485454. URL: https://doi.org/10.14778/3485450.3485454.

[254] Wilson Snyder. *Verilator*. Veripool, 2021. URL: https://www.veripool.org/ftp/verilator_doc.pdf.

[255] *SPEC 2017 FireMarshal Workload*. UCB-BAR. URL: https://github.com/ucb-bar/spec2017-workload.

[256] *SPEC2017*. Standard Performance Evaluation Corporation.
URL: http://spec.org/cpu2017/.

[257] *Spike RISC-V ISA Simulator*. https://github.com/riscv/riscv-isa-sim.
The Regents of the University of California, 2019.

[258] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov.
"Cloudburst: Stateful Functions-as-a-Service".
In: *Proc. VLDB Endow.* 13.12 (July 2020), pp. 2438–2452. ISSN: 2150-8097.
DOI: 10.14778/3407790.3407836.
URL: https://doi.org/10.14778/3407790.3407836.

[259] John A. Stankovic, Krithi Ramamritham, and Marco Spuri.
Deadline Scheduling for Real-Time Systems: Edf and Related Algorithms.
USA: Kluwer Academic Publishers, 1998. ISBN: 0792382692.

[260] Thomas Sterling, Donald J. Becker, Daniel Savarese, John E. Dorband, Udaya A. Ranawake, and Charles V. Packer.
"Beowulf: A Parallel Workstation For Scientific Computation".
In: *In Proceedings of the 24th International Conference on Parallel Processing.*
CRC Press, 1995, pp. 11–14.

[261] Zak Stone. *Now you can train TensorFlow machine learning models faster and at lower cost on Cloud TPU Pods.*
https://cloud.google.com/blog/products/ai-machine-learning/now-you-can-train-ml-models-faster-and-lower-cost-cloud-tpu-pods.
Accessed: 2019-04-12.

[262] Chen Sun, Mark T. Wade, Yunsup Lee, Jason S. Orcutt, Luca Alloatti, Michael S. Georgas, Andrew S. Waterman, Jeffrey M. Shainline, Rimas R Avižienis, Sen Lin, Benjamin R. Moss, Rajesh Kumar, Fabio Pavanello, Amir H. Atabaki, Henry M. Cook, Albert J. Ou, Jonathan C. Leu, Yu-Hsin Chen, Krste Asanović, Rajeev J. Ram, Miloš A. Popović, and Vladimir M. Stojanović.
"Single-chip microprocessor that communicates directly using light".
In: *Nature* 528.7583 (Oct. 2015), pp. 534–538. ISSN: 0028-0836.
URL: http://dx.doi.org/10.1038/nature16454%20http://10.0.4.14/nature16454%20http://www.nature.com/nature/journal/v528/n7583/abs/nature16454.html%7B%5C#%7Dsupplementary-information.

[263] Tuan Ta, Lin Cheng, and Christopher Batten.
"Simulating Multi-Core RISC-V Systems in gem5".
In: *Second Workshop on Computer Architecture Research with RISC-V.*
CARRV '18. Los Angeles, California, USA.

[264] Andrew S Tanenbaum and Robbert Van Renesse. "Distributed operating systems".
In: *ACM Computing Surveys (CSUR)* 17.4 (1985), pp. 419–470.

[265] *The CPU Scheduler in VMware vSphere® 5.1*. Tech. rep. 2013. URL: https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-cpu-sched-performance-white-paper.pdf.

[266] *The Open Group base specifications issue 7*. https://pubs.opengroup.org/onlinepubs/9699919799/. 2018.

[267] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: The next generation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. 2020, pp. 1–14.

[268] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. "Borg: the next generation". In: *Proceedings of the Fifteenth European Conference on Computer Systems*. ACM, Apr. 2020, pp. 1–14. ISBN: 978-1-4503-6882-7. DOI: 10.1145/3342195.3387517. URL: https://dl.acm.org/doi/10.1145/3342195.3387517.

[269] Linus Tordvalds, ed. *Linux Kernel*. 2017. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/.

[270] *Torque Resource Manager*. Version 6.1.3. Adaptive Computing Enterprises, Inc. 2021. URL: https://support.adaptivecomputing.com/wp-content/uploads/2021/02/torqueAdminGuide-6.1.3.pdf.

[271] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. "Disaggregating Persistent Memory and Controlling Them Remotely: An Exploration of Passive Disaggregated Key-Value Stores". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020, pp. 33–48. ISBN: 978-1-939133-14-4. URL: https://www.usenix.org/conference/atc20/presentation/tsai.

[272] *VCS*. Synopsys, Inc. URL: https://www.synopsys.com/verification/simulation/vcs.html.

[273] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. 2015, pp. 1–17.

[274] Jan Vitek and Tomas Kalibera. "Repeatability, Reproducibility, and Rigor in Systems Research". In: *Proceedings of the Ninth ACM International Conference on Embedded Software*. EMSOFT '11. Taipei, Taiwan: Association for Computing Machinery, 2011,

pp. 33–38. ISBN: 9781450307147. DOI: 10.1145/2038642.2038650.
URL: https://doi.org/10.1145/2038642.2038650.

[275] *VMware vSphere Bitfusion User Guide*. Tech. rep. 2021.
URL: https://docs.vmware.com/en/VMware-vSphere-Bitfusion/4.0/vsphere-bitfusion-user-guide-45.pdf.

[276] Werner Vogels. "Web services are not distributed objects".
In: *IEEE Internet computing* 7.6 (2003), pp. 59–66.

[277] Stavros Volos, Djordje Jevdjic, Babak Falsafi, and Boris Grot.
"Fat Caches for Scale-Out Servers". In: *IEEE Micro* 37.2 (Mar. 2017), pp. 90–103.
ISSN: 0272-1732. DOI: 10.1109/MM.2017.32.
URL: https://doi.org/10.1109/MM.2017.32.

[278] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall.
"A note on distributed computing".
In: *International Workshop on Mobile Object Systems*. Springer. 1996, pp. 49–64.

[279] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel.
"The LOCUS distributed operating system".
In: *ACM SIGOPS Operating Systems Review* 17.5 (1983), pp. 49–70.

[280] Qing Wang, Youyou Lu, Erci Xu, Junru Li, Youmin Chen, and Jiwu Shu.
"Concordia: Distributed Shared Memory with In-Network Cache Coherence".
In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*.
USENIX Association, Feb. 2021, pp. 277–292. ISBN: 978-1-939133-20-5.
URL: https://www.usenix.org/conference/fast21/presentation/wang.

[281] Randolph Y. Wang and Thomas E. Anderson.
"xFS: A wide area mass storage file system". In: *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*. IEEE. 1993, pp. 71–78.

[282] William A. Ward, Carrie L. Mahood, and John E. West.
"Scheduling Jobs on Parallel Systems Using a Relaxed Backfill Strategy".
In: *Job Scheduling Strategies for Parallel Processing*.
Ed. by Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn.
Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 88–102.
ISBN: 978-3-540-36180-0.

[283] Andrew Waterman and Krste Asanović, eds.
The RISC-V Instruction Set Manual Volume I: Unprivileged ISA. Version 20191213.
RISC-V Foundation. Dec. 2019.

[284] Andrew Waterman and Krste Asanović, eds.
The RISC-V Instruction Set Manual, Volume II: Privileged Architecture.
Version 20190608-Priv-MSU-Ratified. RISC-V Foundation. June 2019.

[285] Andrew Waterman and Krste Asanović, eds. *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 1.10*. RISC-V Foundation, May 2017.

[286] *Wave Computing Launches the MIPS Open Initiative*. Wave Computing, Dec. 2018. URL: https://wavecomp.ai/wave-computing-launches-the-mips-open-initiative/.

[287] David L. Weaver and Tom Germond, eds. *The SPARC Architecture Manual*. Version 9. SPARC International inc. 1994.

[288] David Wentzlaff and Anant Agarwal. "Factored operating systems (fos) the case for a scalable operating system for multicores". In: *ACM SIGOPS Operating Systems Review* 43.2 (2009), pp. 76–85.

[289] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. "An operating system for multicore and clouds: Mechanisms and implementation". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 3–14.

[290] Bruce Wile. *Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems*. Tech. rep. IBM Systems and Technology Group, Sept. 2014.

[291] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. IEEE Press, 2014, pp. 457–468. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665740.

[292] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. "Anna: A KVS for Any Scale". In: *2018 IEEE 34th International Conference on Data Engineering (ICDE)*. 2018, pp. 401–412. DOI: 10.1109/ICDE.2018.00044.

[293] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, Fan Yang, and Lidong Zhou. "Gandiva: Introspective Cluster Scheduling for Deep Learning". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 595–610. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/xiao.

[294] Xianghua Xu, Peipei Shan, Jian Wan, and Yucheng Jiang. "Performance Evaluation of the CPU Scheduler in XEN". In: *2008 International Symposium on Information Science and Engineering*. Vol. 2. 2008, pp. 68–72. DOI: 10.1109/ISISE.2008.123.

[295] Jian Yang, Joseph Izraelevitz, and Steven Swanson. "Orion: A Distributed File System for Non-Volatile Main Memory and RDMA-Capable Networks". In: *17th USENIX Conference on File and Storage Technologies (FAST 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 221–234. ISBN: 978-1-939133-09-0. URL: https://www.usenix.org/conference/fast19/presentation/yang.

[296] *Yocto Project*. Linux Foundation. URL: https://www.yoctoproject.org/.

[297] Andy B Yoo, Morris A Jette, and Mark Grondona. "Slurm: Simple linux utility for resource management". In: *Workshop on job scheduling strategies for parallel processing*. Springer. 2003, pp. 44–60.

[298] Michael Young, Avadis Tevanian, Richard Rashid, David Golub, Jeffrey Eppinger, Jonathan Chew, William Bolosky, David Black, and Robert Baron. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System". In: *In Proceedings of the 11th ACM Symposium on Operating Systems Principles*. 1987, pp. 63–76.

[299] Peifeng Yu and Mosharaf Chowdhury. "Fine-Grained GPU Sharing Primitives for Deep Learning Applications". In: *Proceedings of Machine Learning and Systems*. Ed. by I. Dhillon, D. Papailiopoulos, and V. Sze. Vol. 2. 2020, pp. 98–111. URL: https://proceedings.mlsys.org/paper/2020/file/f7177163c833dff4b38fc8d2872f1ec6-Paper.pdf.

[300] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. "The datacenter needs an operating system." In: *HotCloud*. 2011.

[301] *ZeBu Server ASIC Emulator*. Synopsys, Inc. URL: https://www.synopsys.com/verification/emulation/zebu-server.html.

[302] *ZeroMQ*. URL: https://zeromq.org/get-started/.

[303] Yiying Zhang, Jian Yang, Amirsaman Memaripour, and Steven Swanson. "Mojim: A Reliable and Highly-Available Non-Volatile Memory System". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015, pp. 3–18. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694370. URL: http://doi.acm.org/10.1145/2694344.2694370.

[304] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. "Sonicboom: The 3rd generation berkeley out-of-order machine". In: *Fourth Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2020.

[305] *Zynq UltraScale+ MPSoC Software Developer Guide*. Version v12.0. Xilinx, Inc. July 2020. URL: https://www.xilinx.com/support/documentation/user_guides/ug1137-zynq-ultrascale-mpsoc-swdev.pdf.