THE UNIVERSITY *of* EDINBURGH

Redesigning Data-Center Supervisory Software for the New Generation of Peripheral Interconnects

Tong Xing

Doctor of Philosophy
Institute of Computing System Architecture
School of Informatics
University of Edinburgh
2024

To my parents

Abstract

We live in an era dominated by cloud computing. Computer systems in the cloud data-center are becoming increasingly powerful but at the cost of complexity. At the forefront of recent technology advancements, newer PCIe generations and the Compute Express Link (CXL) protocol promise significant improvements in connectivity within data-center environments. These innovations introduce a new layer of resource sharing and aggregation, which is crucial for managing the increasing demands of large data volumes and diverse computational processing expected in the near future. However, these advances also increases the complexity of resource sharing in data centers, posing significant challenges to the adequacy of traditional supervisory software designs, including operating systems (OS) and hypervisor.

This thesis explores how to redesign supervisory software for platforms with such emerging high-speed and shared-memory peripheral interconnects, in order to improve application software resource utilization. Specifically, the Thesis investigates how to extend current hypervisor designs to effectively enable CPU oversubscription with PCIe-enabled faster IO, and transparently and effectively use CXL-enabled tiered memory. In addition to that the Thesis explores how to extend traditional OSes to run across multiple server inteconnected by CXL-enabled shared memory, potentially with CPUs of diverse ISA.

The Thesis makes the following contributions. *First,* motivated by the introduction of faster interconnects IO devices, we design and implement Anubis, a new IO-aware VM scheduler for Linux KVM – the most popular Virtual Machine Monitor (VMM) in today's clouds. Anubis is a practical solution that provides close-to-non-overcommit performance for IO workloads in VM overcommitted scenarios. *Second* inspired by the possibility of sharing memory amongst different machines we propose Stramash, a novel operating system kernel design that enables a single-OS among different machines, potentially of diverse ISA. We called such design fused-kernel, which leverages cache-coherent shared memory among heterogeneous ISA CPUs as a fundamental feature. We built a prototype fused-kernel OS, Stramash-Linux, to demonstrate the applicability of our design to monolithic OS kernels. We profile Stramash OS components on real-hardware but tested them on an architectural simulator – Stramash-QEMU, which we design and build. *Third,* driven by the introduction of tiered memory we introduce Libra, an extension of the hypervisor memory subsystem designed to enhance tiered memory management within cloud environments. Libra is a new approach that dynamically adjusts the VM's memory proportion between local and extended memory, optimizing overall memory utilization. *Note,* all solutions have been rigorously evaluated with real-world datasets and have been applied in actual systems to support various applications, and all source code is released open source.

Lay Summary

In today's digital age, cloud computing plays a vital role in powering everything from streaming services to online banking. As our reliance on cloud services grows, so does the complexity of the data centers that support them. These data centers need to handle vast amounts of data and perform complex computations quickly and efficiently.

Recent technological advancements, like the new PCIe 6.0 standard and the Compute Express Link (CXL) protocol, promise to make data centers faster and more connected than ever before. PCIe 6.0 offers much higher speeds for data transfer, while CXL allows different parts of a data center to share memory more effectively. This means that resources like processing power and memory can be shared and used more flexibly, which is crucial for handling the increasing demands on cloud services. However, these advancements also bring new challenges. The software that manages these data centers—like operating systems and virtualization tools—wasn't designed with these new technologies in mind. As a result, they may not use the new hardware as effectively as possible, leading to wasted resources and slower performance.

This thesis explores ways to redesign this supervisory software to better utilize the new high-speed connections and shared memory capabilities. The goal is to make cloud applications run more efficiently by ensuring that the software can fully take advantage of the new hardware. The research makes three key contributions:

Anubis: A new scheduling system for virtual machines that takes into account the faster input/output capabilities of PCIe 6.0. This helps manage computing resources more effectively, even when there are more virtual machines running than there are physical resources (a common situation in cloud environments).

Libra: An enhancement to how memory is shared and managed in virtualized environments. With the introduction of tiered memory systems recently, Libra dynamically adjusts how different tiered memory is allocated to different virtual machines. This ensures that memory is used more efficiently across the data center.

Stramash: A novel approach to operating system design that allows a single operating system to run across multiple servers, even if they have different types of processors. By leveraging shared memory made possible by CXL, Stramash enables better coordination and resource sharing between servers.

All of these solutions have been thoroughly tested with real-world data and are available as open-source software. By rethinking how supervisory software interacts with new hardware technologies, this research aims to make cloud computing more efficient, responsive, and capable of meeting future demands.

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my parents for their patience and understanding—and for believing that my brain was indeed up to the challenge of a PhD.

Then, to my grandparents: I love all of you!

I am immensely grateful to my mentor, Dr. Antonio Barbalace, for guiding me throughout my twenties—the most fascinating years of my life. Without his mentorship, I might never have glimpsed the sanctuary of science. His encouragement and support have fortified my faith and courage, enabling me to face the future with unwavering confidence. Moreover, his humor and personality have profoundly impacted me, making me a better and more confident person.

I would also like to thank my secondary advisor and wonderful neighbor, Dr. Michio Honda, for his invaluable assistance and companionship throughout my PhD journey, including our walks between school and home.

My sincere thanks go to the many professors who have guided me during my studies: Dr. Binoy Ravindran, Dr. Jonathan Balkind, Dr. Pierre Olivier, Dr. Luo Mai, Dr. Boris Grot.

To my dear friends in Edinburgh—Cong Xiong and Yinsicheng Jiang—thank you. Without you, my life here would have been considerably less entertaining.

I am grateful to my ”Systems-Nuts” and ICSA colleagues—the funniest, smartest, and most kernel-loving people at the University of Edinburgh. With the addition of quantum scientists who speak different languages, I continue to enjoy every moment.

I also want to thank my friends in China who have continually supported me. Thank you for the funny stories shared almost every day.

Lastly, I am deeply appreciative of Yanding's support.

Contents

Chapter 1

Introduction

Computing has evolved to become a foundational element of modern civilization, driven by relentless improvements in processing power and reductions in production costs. This progression is encapsulated by Moore's Law, a crucial techno-economic model that has underpinned the IT industry's ability to double the performance and functionality of digital electronics approximately every two years within a fixed cost, power, and area [152] – thanks to advanced electronic design automation tools, compilers, simulators, and emulation technologies. However, within the next decade, the technological basis of Moore's forecast is expected to reach its limit as lithography approaches the atomic scale, as shown in Fig 1.1. Soon, it may be possible to manufacture lithographically produced devices where critical features are only a dozen or so silicon atoms across, representing a practical threshold for the miniaturization of digital computing logic gates.

In response to these looming constraints, the most viable route to sustaining performance enhancements lies not in further improving general-purpose core computing capabilities, but in parallelism and specialized computing. By involving more computing cores, servers have evolved from single multicore systems to novel computing servers that carry hundreds or even thousands of cores. These cores are not only general-purpose processors with different ISA (x86, ARM, Power PC); they can also be specialized computing processors such as ASICs, FPGAs. Further, those computing cores might not be attached to the motherboard directly; they could be connected through the peripheral bus, which forms a distributed computing architecture within a single server. As we step into the era of big data, the data volumes have surged; many applications are now constrained by the performance bottleneck of data movement, which demands additional memory and I/O bandwidth. At the forefront of recent technology advancements, newer PCIe generations and the Compute Express Link (CXL) protocol promise significant improvements in connectivity within data-center environments. The recently introduced PCIe 6.0, as shown in Fig 1.2, has doubled the speed compared to the previous generation, offering higher I/O bandwidth and lower latency. CXL, in particular, is revolutionizing the way that devices connect by enabling shared memory and memory semantic access between connected entities over the peripheral bus. This advancement simplifies resource sharing in the data center, and it is believed to bring higher performance improvement and better utilization (removing fragmentation thanks to the disaggregation). While the exact performance gains remain to be fully

quantified, several studies have discussed this already, for example from Google [118].

These innovations introduce a new layer of resource sharing and aggregation, which is crucial for managing the increasing demands of large data volumes and diverse computational processing expected in the near future. However, these advances also increase the complexity of resource sharing in data centers, posing significant challenges to the adequacy of traditional supervisory software designs, including operating systems (OS) and hypervisors.

This thesis explores how to redesign supervisory software for platforms with such emerging high-speed and shared-memory peripheral interconnects in order to improve application software resource utilization. Specifically, our work investigates how to extend current hypervisor designs to effectively enable CPU oversubscription with PCIe-enabled faster I/O and to transparently and effectively use CXL-enabled tiered memory. Additionally, our work explores how to extend traditional operating systems to run across multiple servers interconnected by CXL-enabled shared memory, potentially with CPUs of diverse ISAs.

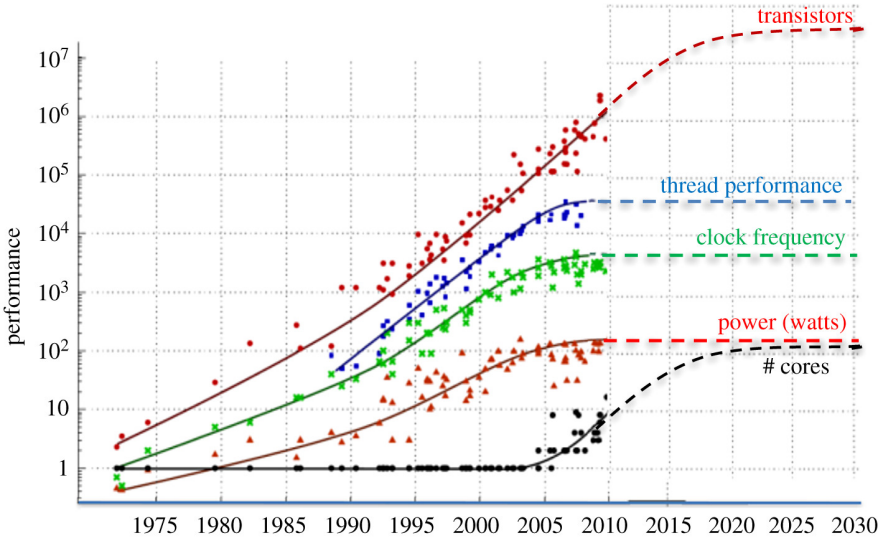
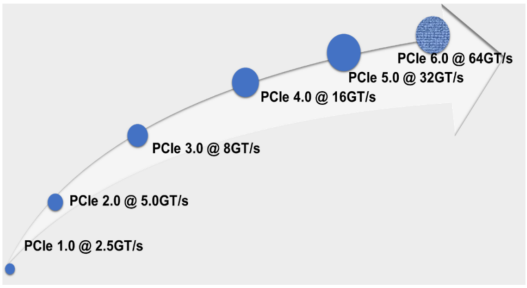


Figure 1.1: All additional approaches to further performance improvements end in approximately 2025 due to the end of the roadmap for improvements to semiconductor lithography, Figure from Kunle Olukotun, Lance Hammond, Herb Sutter, Mark Horowitz and extended by John Shalf

1.1 Motivation

Cloud Computing Today, we live in an era dominated by cloud computing. Cloud computing is the on-demand availability of computing resources (such as storage and infrastructure) as services over the internet. It eliminates the need for individuals and businesses to self-manage physical resources themselves and only pay for what they use. The main cloud computing service models include infrastructure as a service(IaaS) offers compute and storage services; platform as a service, which offers a develop-and-deploy environment to build cloud apps; and software as a service, which delivers apps as services. In the cloud computing environment, computing resources,



PCIe Specification	Data Rate(Gb/s) (Encoding)	x16 B/W per dirn*	Year
1.0	2.5 (8b/10b)	32 Gb/s	2003
2.0	5.0 (8b/10b)	64 Gb/s	2007
3.0	8.0 (128b/130b)	126 Gb/s	2010
4.0	16.0 (128b/130b)	252 Gb/s	2017
5.0	32.0 (128b/130b)	504 Gb/s	2019
6.0	64.0 (PAM-4, Flit)	1024 Gb/s (~1Tb/s)	2021

* - bandwidth after encoding overhead

Figure 1.2: PCIe specification evolution through six generations spanning three decades: Doubling Bandwidth with leading power-efficient and cost-effective performance satisfying the needs across the entire compute continuum as the ubiquitous interconnect, Figure from PCI-SIG consortium

such as CPUs, memory, storage, etc., are organized like an array of elements under the abstraction of virtualization technology. Cloud providers manage these resources, offering virtual machines (VMs) to the client.

In modern data centers, servers are interconnected not only with each other but also with various peripheral hardware devices. To ease of resource management, the cloud provider transverse the cloud platforms to a pooled resource abstraction, which can be precisely tailored to meet specific tenant needs. As a result, virtual machines can be efficiently scheduled and allocated based on the client request. These servers and its peripheral hardware devices are interconnected through the PCIe link, which has continuously evolved to support faster speeds and greater compatibility. A notable advancement in the latest generation of evolution is the PCIe CXL (Compute Express Link) extension. This technology introduced cache coherence between connected entities, facilitating a higher level of resource pooling in the data center. For instance, the tiered memory pool supported by CXL can expand server memory capacity, extending from the local memory of directly attached local DRAM to the remote memory that is connect through the PCIe cable. With cache coherence shared memory, the CXL also allows for more adaptable offloading of computing tasks to peripheral hardware that has SoC on device, optimizing both performance and efficiency in cloud environments.

Multi-tenant Due to the architecture of cloud computing, in order to optimize the utilization of available resources of cloud infrastructure, multiple tenants share the hardware resources and run their workloads together on servers that are across the data center. Because the workloads of the tenants running in the cloud are not always busy, to further optimize resource usage, cloud providers often consolidate multiple workloads onto a single physical server, thereby overcommitting the server's hardware resources – starting from a per-workload declared resource demand upon tenant request, usually estimated as the worst case/peak hardware resources required to meet the target service QoS [41, 73, 76, 133].

However, this hardware resource overcommitting would lead to the resource contention issue in the data center. Which significantly impacts the service performance by introducing extra latency in the service procedure due to the extra waiting time for the available hardware resources, particularly in terms of I/O operations. For instance, the processing of the in/out bound data between the server and peripheral network or

storage hardware devices may suffer from considerably high latency when computing resources are shared and not immediately available. Thus, the new generation of PCIe interface, designed to enhance data transferring and system connectivity, might not reach its full potential performance improvement without proper management of shared hardware resources in the data center. The challenge, therefore, lies in how to enable the current supervisory software, specifically the hypervisor, to efficiently allocate and share hardware resources among multiple tenants without impact to the service performance.

Resource under-utilization A significant challenge in the cloud platforms is the under-utilization of hardware resources. Cloud VMs demand the hardware resource upon tenant request. Therefore, scheduling VMs is transverse to the hardware resources bin-packing problem, which gets more complicated when different constraints in the cloud need to be considered, especially the mixed demands of the current VM and future incoming VM. However, because of the unpredictable arrival and release times of VMs and their variety of hardware resource combinations, it is impossible for cloud providers to fully utilize each server hardware resource. Which leads to resource stranding or spilling in the data center. In this thesis, we focus on the memory resource. The typical scenario for memory stranding is that all cores have been rented, but there is still memory available on the server. The memory spilling is the reverse scenario, which is that cores are remaining, but all memory has been rented.

To reduce memory wasting, cloud providers are trying to disaggregate the memory into a pool that is accessible by multiple servers in the data center. Therefore, breaking the boundary that each server has a fixed hardware resource. By reassigning memory to different servers at runtime, the memory resources can be dynamically adjusted based on demand. Thus, the overall data center's memory-to-CPU ratio can achieve closer to the average VM's memory-to-CPU ratios by tackling the deviations via the memory pool. The recent introduction of the CXL standard technology has paved the way for innovative approaches in data center memory pool management. CXL provides the cache coherence connection between each connected entity and several new hardware have been proposed recently, such as memory expansion module, memory machine, etc. These CXL-enabled memory expansion devices are attached to the server through the PCIe interface, with relatively higher access latency compared to local DRAM. As these new hardware solutions begin to integrate into data centers in the foreseeable future, the new architecture, the tired memory pool, will be put into the data center infrastructure. However, current supervisory software, particularly in cloud computing environments where virtualization is prevalent, struggles to fully exploit these memory resources. The challenge arises from the additional access latency introduced by remote memory, raising critical questions about how to balance memory allocation and utilize both local and remote memory effectively to minimize performance degradation.

Heterogenelty The recent surge in AI/ML specialization has driven the data volume to increase explosively; the traditional Von Neumann design has encountered performance bottlenecks due to the high cost of data movement. To address this, a shift towards near-data processing has been initiated, which strategically places computing

units closer to the data. Therefore, more and more peripheral devices' hardware have embedded the computing core, including TPU, DPU, SmartSSD, SmartNIC, etc. However, most of those embedded computing cores have different ISAs; the introduction of these heterogeneous devices poses a significant challenge: Today's heterogeneous-ISA platforms run a separate software stack per CPU. Applications running thereon are perceived to be in a distributed system and hence cannot leverage per-platform optimizations.

Today, the hardware landscape is rapidly changing. There are several PCIe extensions that consider cache-coherent shared memory (including CXL [78], OpenCAPI [140] and CCIX [70]), making tighter interconnection of heterogeneous processing units an inevitable trend. Therefore, with specialized/heterogeneous computing elements reaching all aspects of our computer systems and their prevalence only growing, we must act to rein in their inherent complexity. One place that has seen significantly less investment in terms of development is heterogeneous-ISA systems, specifically because of such complexity. To date, heterogeneous-ISA processors have required significant software overheads, workarounds, and coordination layers, making the development of more advanced software hard and motivating little further development of more advanced hardware. Given these challenges, a critical question emerges: how to extend traditional OSes to run across multiple servers interconnected by CXL-enabled shared memory, potentially with CPUs of diverse ISA.

Challenges The hardware landscape in cloud computing is changing, marked by increasing connectivity and heterogeneity of hardware resources. As data centers evolve with new hardware in their infrastructure, developing strategies to effectively manage and exploit these faster and highly compatible interconnections remains an open and critical question.

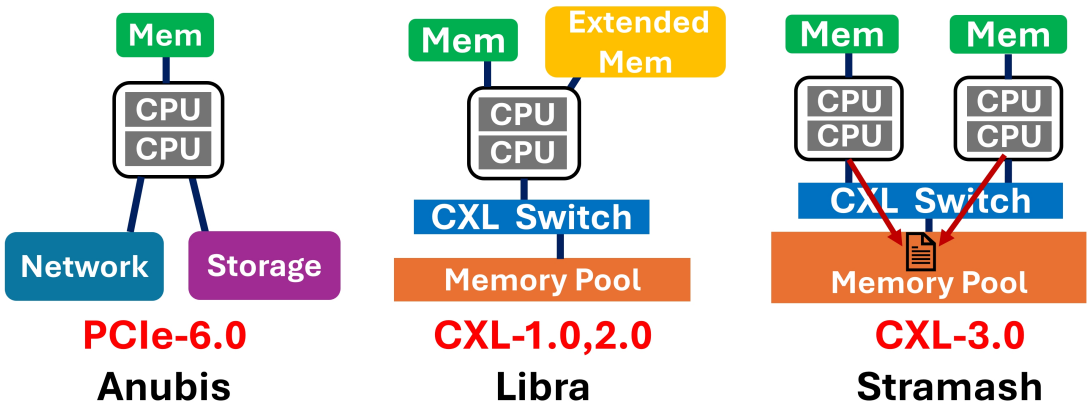


Figure 1.3: From faster interconnected IO devices to the cache coherent shared memory pool

1.2 Contributions

This thesis is made up of the following contributions, which have been developed entirely during the PhD course. Appendix.A lists all works and publications that the student did.

Figure 1.3 illustrates each contribution targeted architecture – from current to the next generation of peripheral interconnections in the cloud. The Thesis is narrated in chronological order of the completion times of the works.

1.2.1 Anubis – Improve I/O Performance in Multi-tenants Cloud

We propose Anubis, a new IO-aware VM scheduler targeting Linux KVM, the most popular VMM in today's Clouds, without requiring any guest VM modifications. Anubis shortens the IO event pending time by lightweight monitoring IO events including interrupt delivery and KVM exit. For the vCPU running the IO activity, Anubis provides an accurate boost, which is exclusively active only during the period when the vCPU has IO activity. While the IO performance is maximized, Anubis still guarantees fairness among VMs. The vCPU that doesn't have IO activity and belongs to the same VM will voluntarily yield the computing resources to counterbalance the unfairness created by the vCPU that has been given a performance boost.

1.2.2 Stramash – Fused Operating System Design

We proposed Stramash; we have introduced a new operating system (OS) design, the fused-kernel OS, which goes beyond the multiple-kernel OS design, exploiting cache-coherent shared memory among heterogeneous ISA CPUs as a first principle – introducing a set of new OS kernel mechanisms. We built a prototype fused-kernel OS, Stramash-Linux, to demonstrate the applicability of our design to monolithic OS kernels. We profile Stramash OS components on real hardware but tested them on an architectural simulator – Stramash-QEMU, which we designed and built.

1.2.3 Libra – Tiered Memory Management in Cloud

We proposed Libra, a system grounded in the insight that Cloud tiered memory management involves an implicit preemption scenario amidst the complexities of a multi-tenant Cloud environment. Libra shifts the focus from page-level profiling to VM-level profiling to select VMs that should receive the memory resources. It introduces a novel concept: a memory scheduler in the Cloud that dynamically adjusts the memory proportion allocated to VMs. Our initial results demonstrate that Libra can eliminate the randomness of memory allocation brought by the page migration and significantly improve local memory utilization, as shown by our simulation results.

1.3 Thesis organization

Chapter 2 introduces the background of Cloud computing, focusing on virtualization technology and the challenges associated with resource contention. We delve into

the role of novel hardware technologies such as Compute Express Link (CXL) and Smart Devices, which are integral to advancing cloud computing capabilities. Additionally, we explore the heterogeneity present in cloud environments, discussing the specific challenges it poses.

Chapter 3 presents the detailed work on Anubis, an IO-aware VM scheduler that extends existing hypervisor designs. The chapter begins by identifying and analyzing the IO challenges that arise when multiple tenants share cloud resources. After investigating the root causes, it presents our design implementation of Anubis. Experimental results are provided to demonstrate the effectiveness of our design and to showcase the improvements Anubis brings to IO handling in multi-tenant environments.

Chapter 4 investigates Fused Kernel design – Stramash, a novel OS design. This chapter starts with the limitations of current heterogeneous OS designs. Then it details the design and implementation of both the Stramash OS and Stramash-QEMU simulator. The experimental section presents the advantages of the fused kernel approach, offering insights and case studies that highlight the benefits of Stramash.

Chapter 5 presents the work related to Libra, a system designed to manage external hardware resources, specifically memory resources. The chapter starts by addressing the limitations of current designs and methodologies for memory resource management. It then introduces the design of Libra and outlines the potential benefits this system offers. The chapter concludes with experimental results that highlight the effectiveness of Libra in managing the tiered memory environment.

Chapter 6 concludes this thesis, summarizing the work presented and discussing the limitations of the contributions, as well as possible directions for future research.

Chapter 2

Background

2.1 Virtualization

Traditional computing architectures often led to underutilization of hardware resources as a single physical machine was deployed for a limited number of applications. In cloud computing environments, hardware resources are typically shared among multiple tenants. Further, data centers also need to support scenarios where tenant workloads require different operating systems (Linux, Windows, Mac OS, etc) or distributions (Ubuntu, Debian, etc). Therefore, to isolate environments and simplify management, cloud providers abstract hardware resources by providing various software layers and granting exclusive access permissions to each tenant. This challenge is addressed through the concept of virtualization, originally pioneered by IBM and later commercialized for x86 computer systems in 1990 [14]. Virtualization abstracts hardware resources into various virtual computing resources, forming the cornerstone of modern data centers. This technique allows for the deployment of multiple virtual servers on a single physical server, thereby enhancing hardware resource utilization. Hypervisor-based virtualization has become a standard approach over the last decade and has been used extensively by major cloud providers. Hypervisors are categorized into two types, illustrate at Fig 2.1:

Type 1 Hypervisors: These operate directly on the hardware with no intermediate operating system, incorporating critical functions such as scheduling within the hypervisor itself. For instance, the Xen [147] hypervisor uses a default scheduling algorithm known as 'credit2' [172].

Type 2 Hypervisors: These run atop an operating system that provides support for devices, memory management, and more. In this case, the hypervisor doesn't include a special scheduler for VMs; instead, each VM's virtual CPU (vCPU) is treated as a regular process on the host and is scheduled by the host operating system's default scheduler. An example is the Linux-based KVM [115].

2.2 Resource contention

Cloud platforms such as AWS, Azure, and GCP often retain a reservation of resource capacity, including CPU cores, memory, and network bandwidth, to ensure scalability

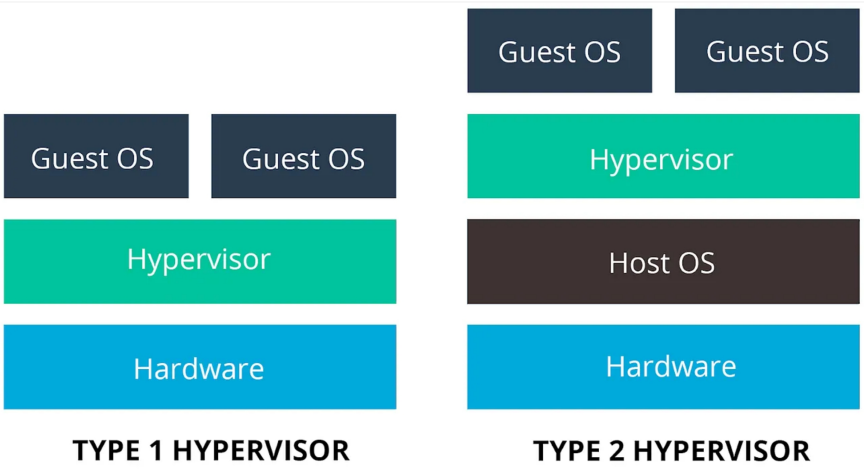


Figure 2.1: Type1 and Type2 Hypervisor structure

and high availability for customer workloads. These platforms deploy customer VMs across various geographical regions and within multiple server clusters. To optimize VM allocation, a bin-packing VM scheduler is used, which assigns VMs to servers based on the availability of resources [119]. Each server has an agent responsible for managing the VM lifecycle, including its creation and termination (including forced eviction to release the resource for higher priority workloads [88]). These measures are part of ongoing efforts to refine processes and enhance the customer experience.

However, resource allocation is complicated by the inherent constraints of cloud VM schedulers and tenants' tendencies to request more resources than necessary – to buffer against potential usage peaks. This overprovisioning makes it difficult to tightly pack VMs, particularly when the DRAM-to-core ratio of incoming VM requests does not match the available server resources. To address these challenges, cloud providers consolidate multiple workloads on a single physical server, overcommitting servers' hardware resources – starting from a per-workload declared resources demand, usually estimated as the worst case/peak hardware resources.

This consolidation involves various resource types such as CPU, memory, network/disk I/O, etc. For example, AWS introduced Burstable VMs [53], which provides a baseline CPU performance with the ability to burst above the baseline at any time for as long as required, similar VM types including Spot Instances [52] and Harvest VMs [42]. Meanwhile, VM Memory Ballooning [18] is used to dynamically adjust VM memory capacity, and Azure has recently proposed a research prototype called Memory Harvest VMs [88], which can further exploit spare capacity by harvesting memory resources. Additionally, several technologies have been adopted to isolate I/O resources among VMs, for example, Storage IO Control(SIOC) from vSphere [34].

When hardware resource contention occurs, the hypervisor may, depending on its policies, either slow down all involved workloads or evict lower-priority workloads to alleviate resource pressure. However, such trade-offs and their impacts on tenants are not well understood. Therefore, future hypervisors aim to meet specific Service Level Agreements (SLAs) and achieve fair sharing of hardware resources among different VMs, balancing resource allocation to maintain service quality without compromising system stability.

2.3 Hardware

The Peripheral Component Interconnect (PCI) [23] is a local computer bus standard that was introduced in the early 1990s to replace older bus standards like ISA [17] and VESA Local Bus [33]. It was designed to facilitate the attachment of peripheral devices to a computer's motherboard, enabling components such as network cards, sound cards, modems, and disk controllers to communicate with the CPU and memory. PCI provided a shared parallel bus architecture, allowing multiple devices to connect and operate concurrently. However, as computing demands grew and the need for higher bandwidth and faster data transfer rates increased, the limitations of the PCI architecture—such as its lower bandwidth capacity and scalability issues—became apparent.

To address these limitations, the industry developed PCI Express (PCIe) [24], which serves as the modern evolution of the PCI standard. For over two decades, the PCIe architecture has been the ubiquitous backbone interconnect in the computing landscape, serving as a fundamental component in modern data centers. As shown in Fig2.2, the PCIe standard has evolved significantly over its generations.

PCIe features a layered architecture in which each layer provides distinct, well-defined functionality, allowing for independent evolution and development. This modular design is similar to that of conventional PCI but introduces several advanced features: **Integration with the System:** PCIe devices are tightly coupled to the systems they are connected to, ensuring seamless communication and coordination. **Direct Memory Access:** PCIe devices can directly access the system's memory using standard memory read and write operations, enabling efficient data transfers without burdening the CPU. **System Address Map Integration:** The system's address map includes any device memory exposed by the PCIe device. This memory is accessible to other devices within the system, including CPUs, using standard memory semantics.

The recently introduced PCIe 6.0 specification [25] doubles the bandwidth of its predecessor, PCIe 5.0 [178] (which operates at 32 GT/s), while continuing to meet industry demands for a high-speed, low-latency interconnect. PCIe 6.0 technology promises to provide a cost-effective and scalable interconnect for modern data centers, offering high bandwidth and low-latency connections essential for data-intensive applications such as artificial intelligence and machine learning (AI/ML), high-performance computing (HPC), automotive systems, and more.

2.3.1 CXL

Compute Express Link® (CXL®) is an open standard interconnect for high-speed, high-capacity CPU-to-device and CPU-to-memory connections, designed for high-performance data center computers [75, 78]. CXL is built on the serial PCI Express (PCIe) physical and electrical interface and includes PCIe-based block input/output protocol (CXL.io) and new cache-coherent protocols for accessing system memory (CXL.cache) and device memory (CXL.mem).

CXL technology maintains memory coherency between the CPU's memory space and the memory on attached devices, allowing for resource sharing that leads to higher performance, reduced software stack complexity, and lower overall system cost. This enables users to focus on target workloads rather than dealing with redundant memory management hardware in their accelerators. CXL is intended to be an open indus-

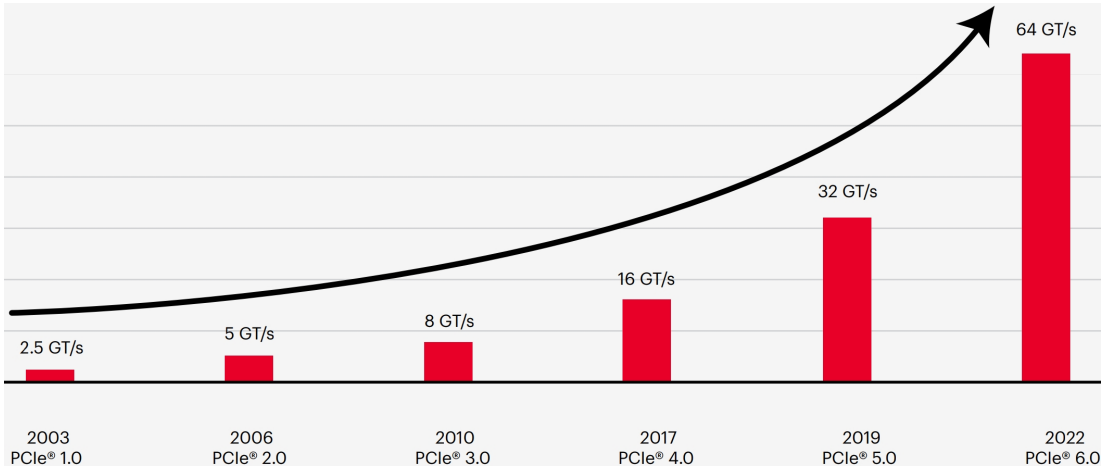


Figure 2.2: Evolution of the PCIe standard over generations, Figure from PCI-SIG consortium

try standard interface for high-speed communications, especially as accelerators are increasingly used to complement CPUs in emerging applications such as artificial intelligence and machine learning.

CXL defines a broad array of devices within its interconnect framework, encompassing traditional and specialized hardware such as graphics processing units (GPUs), general-purpose GPUs (GPGPUs), field-programmable gate arrays (FPGAs), and a wide range of special-purpose accelerators in networking and storage I/O devices. For example, CXL facilitates the integration of memory devices by allowing them to connect to the CPU over the PCIe interface – CXL Memory Module (CMM) [102] – which is conventionally linked to the CPU through the Double Data Rate (DDR) memory interface. This enhancement extends the interface capabilities between CPUs and memory resources, enabling features like memory expansion, pooling, and sharing. Fig2.3 illustrates the architecture of the latest prototype product—a memory machine—which is a memory expander controller that manages all attached DRAM and provides a management interface to the connected server through the CXL link.

2.3.2 Near data processing

In the era of big data, computer systems are experiencing unprecedented volumes of data. Large corporations like Facebook have stored over 300 petabytes (PB) of data in their warehouses, with an incoming daily data rate of 600 terabytes (TB) as of 2014 [30]. Recent warehouse-scale profiling shows that data analytics has become a major workload in data centers [108]. Operating on such a scale of data presents a significant challenge for system designers. Therefore, designing efficient systems for massive data analytics has increasingly become a topic of major importance.

One innovative approach to addressing these challenges is near-data processing. Instead of moving large volumes of data from the device to the host—a process that can become a significant bottleneck—the idea is to place the computing core closer to the device that contains the data. By processing data directly where it is stored, systems can avoid unnecessary data movement and improve overall efficiency.

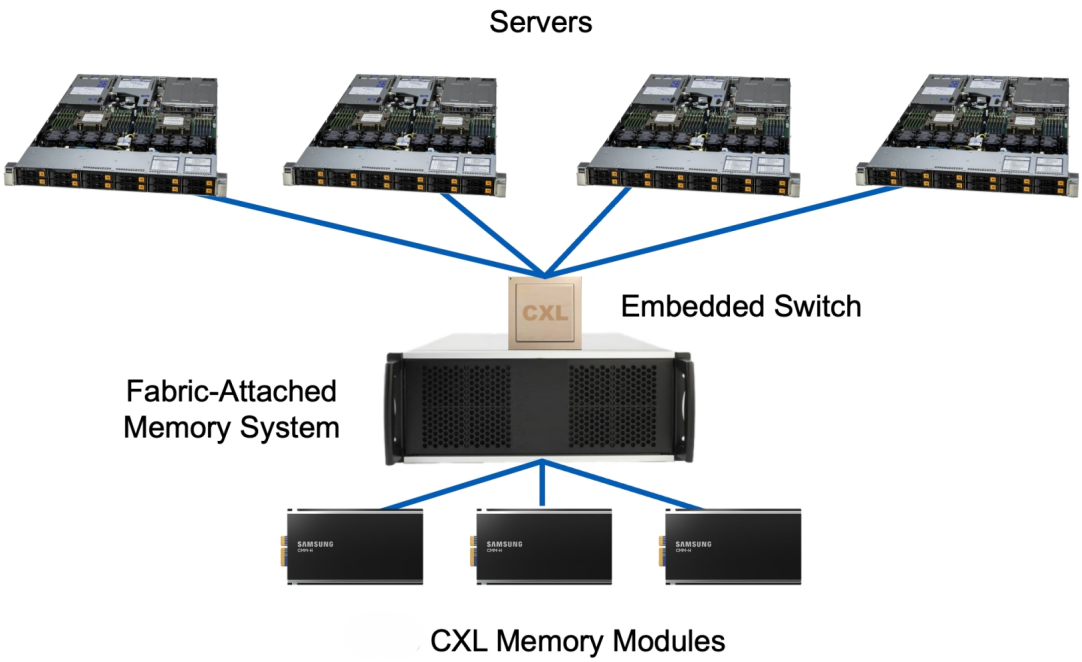


Figure 2.3: Overview of Memory Machine™ architecture, Figure from MemVerge [20]

Guided by this insight, existing work has leveraged embedded ARM cores, FPGAs, or ASICs for computing task offloading. A ”smart device” is a technology where a processing unit is integrated into a traditional peripheral device, such as a solid-state drive (SSD) [29], network interface card (NIC) [7], or an in-memory computing unit like UPMEM [32]. This processing unit can be programmable, like a multicore CPU, or reconfigurable, like an FPGA. Adding processing capabilities to these conventional I/O devices offers several advantages, including early data processing and offloading computation from the host processor to the device's computing core. This not only reduces latency but also improves the system's ability to handle large-scale data analytics efficiently.

2.4 Heterogenelty in Cloud

The incorporation of servers and devices with heterogeneous Instruction Set Architectures (ISAs) introduces significant challenges in data center operations. For instance, AWS recently introduced Graviton[38], a powerful ARM-based CPU designed to offer optimal price-performance for cloud workloads on Amazon EC2 [1]. Cloud providers frequently need to partition their data centers based on ISAs or statically assign workloads to machines with specific ISAs. The common practice involves creating separate ISA partitions—specifically ARM and x86 zones in OpenStack[22]—which can lead to inefficiencies in workload allocation. One partition might become overloaded while another remains idle, resulting in wasted computing resources and potential service disruptions. Therefore, there is a pressing need for modern data centers to enable workloads to migrate across different ISAs, enhancing resource utilization and service robustness.

Native applications, although executable on heterogeneous ISA servers, cannot migrate between them at runtime. In contrast, applications developed in retargetable or intermediate languages like Java or Python can run on heterogeneous platforms but are typically statically assigned to specific servers upon creation. While tools exist to facilitate runtime migration for these languages, migrating stateful applications remains costly, mainly due to the serialization and deserialization processes required between different ISA-specific formats. Moreover, many applications are developed in lower-level languages like C for efficiency—for example, the Redis server [149]. This underscores the critical need for cross-ISA migration to improve energy proportionality and overall energy efficiency in data centers. Such capabilities would allow for more dynamic resource management and potentially lower energy consumption by enabling the seamless reallocation of workloads across different processor architectures.

2.5 Multi-kernel OS

The multikernel model redefines traditional operating system architectures by structuring the OS as a distributed system of cores that communicate solely through messages, with no shared memory between the cores except for messaging channels. This design approach is based on the premise that all inter-core communication within the OS is conducted using explicit messages. The multikernel method becomes increasingly efficient compared to shared-memory access, particularly as the number of cache lines involved grows—a trend expected to become more pronounced in the future [61, 62, 179].

It is important to note that while the multikernel design does not inherently rely on shared memory, it does not prevent applications from sharing memory between cores for their internal needs. A fundamental aspect of the multikernel approach is its abstraction from the hardware. The design minimizes the OS's dependency on specific machine architectures, focusing primarily on two areas: the messaging transport mechanisms and the interface to hardware such as CPUs and devices. This separation offers several key advantages, including enhanced flexibility and scalability.

In traditional operating systems, state management—such as the Windows dispatcher database or Linux scheduler queues—requires shared data structures protected by locks. However, in a multikernel environment, the explicit lack of shared memory necessitates a model where the global OS state is replicated across cores. This replication is facilitated through explicit communication, transforming how state is managed and accessed across multiple processors.

Despite its theoretical elegance and the clean separation of concerns it offers, the multikernel model represents an idealistic approach to OS design. It views the operating system as a fully distributed system with no shared state, which introduces several practical implications and challenges in a real-world context.

Recently, emerging technologies like the CXL standard have enabled connected entities to access each other through cache-coherent shared memory. This advancement facilitates tighter interconnections and calls into question the traditional multikernel, shared-nothing model. It prompts us to consider whether some elements can be shared between multiple kernels to fill the spectrum of operating system design, as illustrated in Fig 2.4.

Figure 2.4: Spectrum of distributed OS design

2.6 Cross-ISA migration

In the late 20th century, numerous projects [46, 80, 159] explored the migration of applications across heterogeneous machines by converting entire application states, including data, from one ISA format to another. This process, however, was associated with significant overheads due to the extensive state conversion required. Recent advancements have significantly improved upon these methods. Notable among these are HSA, Venkat et al., and Popcorn Linux[59, 151, 166], which have introduced a common data format across different ISAs, substantially reducing the amount of state conversion necessary.

Both Popcorn Linux and Venkat et al. propose a uniform address space layout and a common data format among general-purpose CPUs of different ISAs. This approach assumes that the ISAs support identical sizes and alignment constraints for primitive data types, as well as consistent endianness. To facilitate valid pointer functionality across migrations, every program symbol—including functions and variables—must reside at the same virtual address for each ISA. As different ISAs cannot execute the same machine code, each program function or procedure is compiled into the machine code of every ISA. These compiled versions are stored at identical virtual addresses, resulting in overlapping `.text` sections in the virtual address space.

Due to this uniform address space layout and common data format, migrating a thread or process between ISAs is almost as straightforward as migrating a thread between different CPUs on a symmetric multiprocessing (SMP) machine, where there is no need for state transformation. However, since CPUs of different ISAs have varied register sets, some state transformation remains necessary. Prior works have addressed this by converting the register states between architectures. Popcorn Linux, for example, performs this conversion within the kernel space. Moreover, migrations between CPUs of different ISAs can only occur at designated migration points—machine instructions where it is feasible to transform the architecture-specific state, such as registers.

Additionally, while the register state undergoes conversion, each thread's stack remains in the native format of the original ISA and is converted upon migration. Popcorn Linux incorporates code into its compiler – Popcorn Compiler [11, 26, 27] – that automatically handles the stack conversion during the migration process.

Chapter 3

Overcommiting vCPU Scheduling

3.1 Introduction

Many workloads running in the data center do not always require peak stage hardware resources [79]. Thus, major cloud providers consolidate multiple workloads together, on a single physical server, and oversubscribe servers' hardware resources – starting from each workload's declared hardware resources requirement upon creation, usually estimated as the lowest/peak stage hardware resources demand [41, 73, 76, 133]. In this thesis, we focus on *CPU resources*, and look at *multiple-CPU Virtual Machines* (VMs) as single-tenant workload carriers. When consolidating multiple VMs on the same physical server, while oversubscribing resources, the virtual CPUs (vCPUs) of different VMs are going to process on the same physical CPU (pCPU). The supervisory software, the hypervisor (or Virtual Machine Monitor, VMM) schedule time-multiplexes different vCPUs on each pCPU, trying either to reach a specific SLA, or to achieve the fair sharing of a pCPU that carries multiple different vCPUs, and balancing the load among pCPUs [65]. Herein, we focus on the latter – the *fair scheduling* [65, 111] of *oversubscribed VMs*, unfortunately, due to the nature of virtualization, the hypervisor scheduler and the software running in the VM have a semantic gap in between, which often creates VM IO workload underperformance, especially for latency-sensitive IO workloads. With the new generation of PCIe interface integrated into the modern data center, the hardware stack overhead of the IO process is kept decreased, which amplifies the overhead of the software stack overhead. Therefore, Anubis is trying to eliminate the multi-tenant interference on IO workloads due to the CPU resource contention and achieve close to non-oversubscribed native performance.

Problem. Because by oversubscribing resources, vCPUs of different VMs share the schedule `runqueue` of pCPU. When a vCPU is scheduled, the tasks within it become active and are processed. The tasks of a vCPU run only when that vCPU is running: when a vCPU exhausts its *time slice*, it is preempted and paused – i.e., inactive, causing also its tasks to pause execution. **Fig. 3.1** compares the scenario of oversubscription (2 vCPUs on 1 pCPU) with non-oversubscription – IO events cannot be delivered when a vCPU is inactive, and this can lead to several problems, particularly for IO-related workloads:

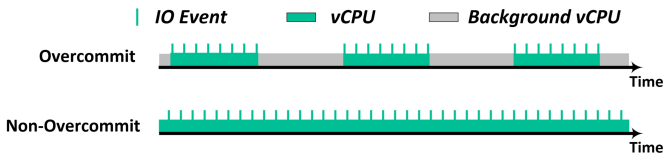


Figure 3.1: *VM oversubscribed impact on IO Event*

- as the vCPU is paused, all interrupts, including the IO-related interrupt from IO devices(hardware interrupt) and inter-processor-interrupts (IPIs)(software interrupt), cannot be processed upon receipt. Therefore, IO-related workloads will stay pause until the interrupt is being handled;
- the vCPU could be rescheduled during the IO task processing, which will force the processing to pause and lead to substantial tail latency for the IO-related service;
- the throughput of IO-related tasks could be affected because fewer IO requests are handled per unit of time, this results in fewer responses being generated, which in turn causes even fewer requests in the subsequent round.

How to lift the impact when vCPUs become inactive? In other words, could the performance of VMs' IO tasks in an oversubscribed scenario be as good as their performance in a non-oversubscribed scenario?

Current Solutions. The problem of *IO performance degradation due to non-active vCPUs in an oversubscribed scenario* is well-studied in the field of systems research [71, 82, 83, 90, 106, 107, 109, 114, 124, 125, 139, 156, 160, 162, 174, 175]. Existing most related research mainly adopts one of two ideas:

- Modifying the VM scheduler to shorten the non-active period of vCPU (vSlicer [174], vBalancer [71], AQL_sched [162]);
- Tracing IO workloads within the VM to make sure the vCPU will not be rescheduled during IO task processing (xBalloon [160], vMigrater [107], partial-boost [114]).

Those approaches have different limitations detailed in **§ 3.3**. For instance, the leading method in this area, vMigrater, requires modifications to the guest VM, thus lacking transparency and offering no legacy support, which significantly restricts its immediate applicability in cloud environments. Additionally, vMigrater has proven to be effective for VMs equipped with a large number of vCPUs, yet it shows reduced efficacy for VMs with fewer vCPUs (e.g., 1, 2, 4). However, it is documented that over 86% of VMs in the cloud utilize 4 vCPUs or fewer [95], indicating that vMigrater is not a universal solution. Our approach is informed by these previous works but goes beyond merely amalgamating their features. Rather than just minimizing vCPU inactivity time, we aim to boost the responsiveness of vCPUs by prioritizing VMs upon receiving an IO event. Additionally, we advocate for a paradigm shift from prioritizing the IO task within the vCPU to prioritizing the vCPU itself that handles the IO tasks.

Anubis. We observe that if an application performing IO were running directly on an OS on bare-metal, rather than in a VM, the OS scheduler would *wake up* wake up the application waiting for IO soon after the IO interrupt is received. This led us

to question: *can the hypervisor* identify the IO events *occurring in a VM and* change the vCPU's scheduling *decisions accordingly?* For example, by immediately waking up a non-running vCPU when it receives an IO interrupt to enable low-latency IO processing. To optimize IO performance, it is crucial to give priority to the vCPU handling IO tasks, ensuring it is not preempted. Thus, we ask: *can we* accurately identify *the start and end points of an IO event within a vCPU?* While this strategy promises enhanced IO performance, it might lead to potential unfairness among co-executing VMs, which could suffer from a reduced share of resources. Therefore, we also need to consider: *can the new scheduler* conserve fairness*?*

We designed and implemented *Anubis* to address the questions raised earlier. *Anubis* is a novel IO-aware VM scheduler that builds upon the traditional fair scheduling algorithms [65] utilized in contemporary Operating Systems and hypervisors. As such, it aims to maintain equitable scheduling decisions across VMs. To minimize vCPU inactivity, *Anubis* carefully monitors the delivery of interrupts from the hypervisor, ensuring that each interrupt is processed promptly upon delivery. Although this approach might increase the number of context switches, our evaluation shows that the associated overhead is minimal. To preserve the priority of vCPUs engaged in IO operations, *Anubis* employs lightweight VM introspection techniques to monitor IO events on a per-vCPU basis, without requiring any modifications to guest VM software. This precise detection of IO events enables *Anubis* to temporarily boost the priority of vCPUs specifically during these events. Furthermore, to uphold fairness in VM scheduling, *Anubis* introduces a debt-like system. This system ensures that any temporary prioritization adjustments for IO processing do not lead to long-term disparities in resource allocation among co-hosted VMs.

Anubis is implemented within the Linux KVM hypervisor framework. It enhances the Linux kernel's Completely Fair Scheduler (CFS) and integrates with Linux's KVM subsystem to effectively manage VM resources. *Anubis* has undergone comprehensive evaluation, particularly in oversubscribed scenarios. It consistently enhances IO performance, bringing it close to levels observed in non-oversubscribed scenarios. Consequently, this significantly reduces the latency experienced by IO-intensive applications, streamlining their operation and improving overall efficiency.

Contributions. We make the following key contributions in this work:

- *First*, we explore how fair scheduling of VMs significantly influences the IO performance of oversubscribed VMs and analyze the shortcomings of previous works in providing deployable solutions for Cloud providers.
- *Second*, we introduce *Anubis*, a novel approach that enhances IO performance while maintaining fairness. *Anubis* does not necessitate any modifications to guest VMs and is structured around three principal strategies: mitigating the impact of vCPU inactivity, maximizing IO performance accurately, and upholding overall fairness, each supported by innovative mechanisms and policies.
- *Third*, we have developed and thoroughly evaluated a prototype of *Anubis* based on a recent version of the Linux KVM. This evaluation demonstrates that *Anubis* is straightforward to implement and effectively boosts IO performance while ensuring fairness.

The remainder of this chapter is organized as follows. **§ 3.2** introduces the background and motivation of Anubis. **§ 3.3** introduces the previous works. **§ 3.4** presents the design principles and details of the Anubis. **§ 3.5** describes implementation details.

§ 3.6 presents the evaluation of the Anubis, and **§ 3.7** concludes.

3.2 Background & Motivation

I/O performance issues stemming from vCPU inactivity have been extensively explored, yet despite numerous proposed solutions, integration into production environments has been limited due to inherent drawbacks. Notably, the approach in [114] focuses exclusively on single-CPU VMs, which, while common, are now less prevalent than multi-CPU VMs [95]. Concurrently, more recent efforts aimed at improving multi-CPU VMs [107, 160] necessitate modifications to both user- and kernel-level software within the guest VMs. This requirement complicates their implementation in cloud environments, posing significant challenges for widespread adoption.

3.2.1 Hypervisor

Hypervisors, or Virtual Machine Monitors (VMMs), are software, firmware, or hardware that create and manage virtual machines. There are mainly two types of VMMs: *type 1*, like Xen [147], and *type 2*, like Linux KVM [115]. Type 1 VMMs run directly on the bare-metal – without any operating system or runtime in between. The scheduler is part of the VMM itself. For example, the Xen hypervisor implements its own scheduling algorithm(s), being *credit2* the default [172]. Type 2 VMMs run atop an operating system, the operating system provides support for devices, memory management, etc. The operating system scheduler does schedule the VMs together with processes and threads – i.e., the VMM does not implement a scheduler. Hence, a Linux KVM virtual machine is likely scheduled by Linux's fair scheduling algorithm (details in **§ 3.2.2**).

Similar to the fair scheduler used by Linux KVM, Xen's credit2 scheduler also aims to maintain fairness among running vCPUs. In both systems, a vCPU engaged in an IO task typically receives a higher priority due to its reduced CPU usage while awaiting IO events.

In the credit2 system, a vCPU occupied with an IO task accumulates credits, and those with higher credits are given scheduling preference, allowing them to preempt other vCPUs. However, if a vCPU performs both IO and CPU-bound tasks, it will not receive this prioritization under credit2. This is because the vCPU expends its accrued credits on CPU-bound activities, rather than conserving them during IO waits. For more detailed information, see **§ 3.2.4**.

Currently, *Anubis* is primarily focused on type 2 VMMs due to their widespread use, as documented in [143, 180]. Specifically, Linux KVM is gaining significant momentum in cloud environments, with recent research in virtualization increasingly favoring Linux KVM over Xen, as discussed in **§ 3.3**. However, the principles underlying *Anubis* could also be adapted for type 1 VMMs, particularly Xen. This adaptation would involve substituting KVM's `vruntime` mechanism with Xen's credit quantum to integrate *Anubis*'s functionalities.

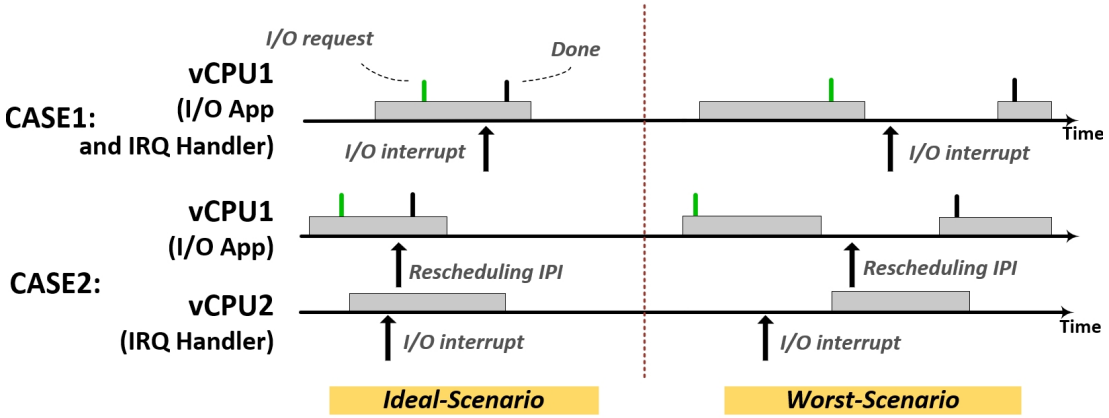


Figure 3.2: *Ideal and worst scenarios of two cases in which vCPU scheduling may affect IO performance.*

3.2.2 Fair Scheduling

Linux employs the CFS algorithm by default to schedule tasks, whether they are threads or processes. The fundamental aim of CFS is to ensure a fair distribution of CPU time among all threads, promoting a balanced system. The primary metric used by CFS to determine scheduling priority is the ”virtual runtime” (`vruntime`), which tracks the amount of CPU time a process has consumed. Under CFS, the thread with the lowest `vruntime` is considered the highest priority and is therefore selected to run next.

IO tasks, which typically spend much of their time sleeping while waiting for IO events, tend to accumulate less `vruntime`. Consequently, they often have a lower `vruntime` compared to CPU-bound tasks, giving them higher priority in the scheduling queue. Therefore, when an IO task wakes up, it is likely to preempt currently running tasks, quickly taking over CPU resources to handle IO processing.

3.2.3 Interrupt Handling

The fair scheduling of different vCPUs equally splits the pCPU time among vCPUs, and traditionally the VMM scheduler doesn't know if a guest software is waiting for IO, or producing IO, etc. Hence, if an IO event for an application is received by the VMM just after the target VM's vCPU has been preempted in order to make another vCPU run, such event (likely an interrupt) will be delivered to the target VM's vCPU at its next timeslice, which can introduce a significant delay to the event processing and degrade performance. This is depicted as **CASE 1** in **Fig. 3.2**. **CASE 2** in the same Figure shows a more unfortunate scenario where two timeslices need to elapse before an IO event is delivered to the application. When a VM with multiple vCPUs receives an interrupt on a vCPU that was not supposed to process it – because the application requesting the IO is on another vCPU, the receiving vCPU will generate a rescheduling IPI to the vCPU with the application. However, the rescheduling IPI may be sent to a vCPU that has just been preempted – thus, a timeslice should pass before such IPI is processed. Moreover, the actual interrupt may be delivered to a just preempted vCPU, adding another timeslice to be waited for, in order to deliver the IO event to the application in the VM.

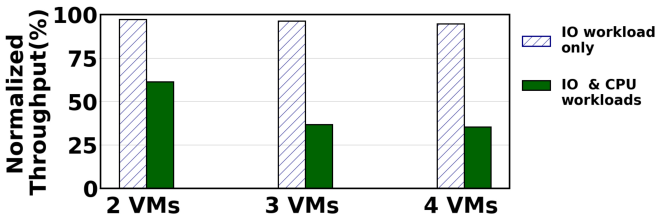


Figure 3.3: *sysbench-seqrd [161] running without (IO Workload only) and with CPU-bound workload (IO & CPU workload) results normalized to the non-oversubscribed case. All VMs with 4 vCPUs, oversubscribe ratios 2/3/4:1.*

3.2.4 Semantic Gap

In an oversubscribed environment, multiple vCPUs share the same pCPU. In this setup, the vCPU with the lowest `vruntime` receives the highest priority in Linux KVM. To the host's CFS, a vCPU is treated simply as another task or thread. Because of this semantic gap, the CFS does not know which applications are running inside the vCPU.

If a vCPU that is running an IO workload shares a pCPU with other vCPUs running a CPU-bound workload, the host CFS effectively manages this setup, maintaining the same IO performance as observed in the non-oversubscribed scenario. This is demonstrated in the **IO workload only** bars in **Fig. 3.3**. Because the IO thread will idle while waiting for an IO event, the vCPU it occupies will also idle, resulting in lower accumulated `vruntime`. When the IO event for this thread arrives, the host CFS quickly schedules this vCPU for execution. However, the situation changes when a vCPU handles both IO and CPU-bound workloads, as illustrated in the **IO & CPU workloads** bars in **Fig. 3.3**. Since the CPU-bound workload continues running when the IO workload is idling, the vCPU does not relinquish control of the pCPU, leading to a higher accumulation of `vruntime`—similar to what is seen with vCPUs that are solely CPU-bound. This accumulation causes the mixed workload vCPU to be deprioritized, potentially delaying the processing of interrupts and the handling of IO events.

Essentially, the host fair scheduler views a vCPU running both IO and CPU-bound tasks in the same light as a vCPU dedicated solely to CPU-bound tasks, with both types accruing the same amount of `vruntime` in each scheduling period. This behavior is consistent with what is observed in Xen's credit2 system. As a result, vCPUs with mixed workloads may not achieve low IO latency under the current VM fair scheduling mechanisms. This highlights the need for an innovative IO-aware VM fair scheduler designed to promptly preempt non-IO tasks while preventing the preemption of IO-bound tasks, thereby improving overall performance for IO-heavy workloads.

3.2.5 Target Problems

When multiple VMs hosting purely CPU-bound workloads are oversubscribed, their overall performance degrades because they share limited CPU resources. *Anubis* is not intended to handle this computing-only scenario. Instead, *Anubis* addresses cases where a VM handles both IO-bound and CPU-bound tasks.

To illustrate the problems *Anubis* aims to address, consider two basic examples: Linux's `ping` command and `sysbench-seqrd` [161]. In an oversubscribed environment, a target vCPU that executes these commands may share a pCPU with another

Table 3.1: *Latency results of* `ping` *and* `seqrd` *when running in a VM with or without additional CPU-bound workload. In the oversubscribed case an additional CPU-bound VM runs on the same pCPU(s).*

Type	**Background pressure**	**Latency(ms)**
ping *(non-oversubscribe)*	1%	0.253
ping + CPU-bound *(non-oversubscribe)*	100%	0.252
ping *(oversubscribe)*	1%	0.286
ping + CPU-bound *(oversubscribe)*	100%	**2.159**
seqrd *(non-oversubscribe)*	2%	0.367
seqrd + CPU-bound *(non-oversubscribe)*	100%	0.367
seqrd *(oversubscribe)*	2%	0.374
seqrd + CPU-bound *(oversubscribe)*	100%	**7.167**

vCPU dedicated solely to a CPU-bound workload. This setup places additional computational pressure on the target vCPU, affecting its ability to efficiently handle both `ping` and `sysbench-seqrd`, regardless of whether it is running these tasks alone or in conjunction with other CPU-bound workloads.

As shown in **Table 3.1**, in the non-oversubscribe case or the oversubscribed case without CPU-bound workload, the CFS scheduler can schedule the vCPU properly: the latency remains low. However, in the oversubscribed scenario where the vCPU runs an IO and a CPU-bound task, the performance of the IO task is *heavily affected*: 8x and 19x longer tail latency.

3.2.6 Target Workloads in Cloud

Anubis primarily aims at spot instances where workloads have lower durability or QoS requirements (e.g., throughput). Clients on spot instances typically recognize that CPU resources may be shared, but do not anticipate the magnitude by which IO performance can degrade—particularly in terms of latency. For example, our measurements on both Huawei Cloud and AWS show an average ping latency increase of 800%, with the 99th percentile latency skyrocketing by up to 4000%.

While purely IO-bound workloads with minimal CPU usage can sometimes avoid severe performance degradation – since the host CFS scheduler can quickly preempt for IO – the reality is that most cloud applications handling IO also involve some amount of computation. Examples of workloads commonly found in cloud environments include live streaming transcoding, web servers, faas, storage services, and email servers [47, 48, 49, 50, 51, 164]. These applications often involve computation alongside their IO operations, both of which are latency-sensitive. These mixed IO+CPU workloads are exactly what *Anubis* seeks to protect against the pitfalls of oversubscription.

3.3 Analysis of Previous Works

Previous works proposed various solutions to mitigate the IO performance degradation due to vCPU inactivity in oversubscribed scenarios. We organize such works into 3 categories, each of which is discussed below. **Table 3.2** summarizes previous works, highlighting what software modifications they require, what VMM they target, and their key limitations.

3.3.1 vCPU Inactivity Period Reduction

vSlicer extends Xen's *credit* scheduler [172], a precursor to *credit2*. vSlicer [174] enhances vCPU responsiveness by increasing the rate of task context switches during IO processing, effectively shortening the scheduling timeslice. This adjustment successfully lowers IO latencies even in oversubscribed VM environments. As a result, tail latency is reduced due to the shortened inactivity periods of the vCPU. However, the vCPU remains susceptible to preemption at any time, which can worsen the worst-case latency, as discussed in **§ 3.2.3**. Consequently, it might take two timeslices before an interrupt is handled. Additionally, the reduction in vCPU inactivity periods increases the frequency of costly VM context switches, which diminishes the total workload that VMs can handle within a given time unit.

Similarly, AQL_sched [162] modifies the scheduling intervals of vCPUs to enhance performance. It profiles and categorizes each vCPU based on the workload it runs, assigning an appropriate "quantum length" according to the categorized workload type. However, this quantum length is preset and static, presenting a limitation if a vCPU changes its workload type during runtime, as AQL_sched lacks the capability to dynamically adjust the quantum length. Should all vCPUs be set to operate with the shortest quantum length, AQL_sched would encounter the same issues as vSlicer, suffering from increased context switching and related inefficiencies. Therefore, both AQL_sched and vSlicer are subject to similar challenges in managing vCPU performance effectively.

Another approach is vBalancer [71], which tries to deliver IO device interrupts only to the running vCPU(s) by redirecting interrupts at the hypervisor level. This approach can reduce the pending time of interrupt handling. In other words, it reduces the VM's interrupt response time as other non-inactive vCPUs can assist in processing interrupts quickly. However, this approach is not aware of the rescheduling IPI delivery problem mentioned in **§ 3.2.3**, which may lead to no interrupt response time reduction even if interrupts are redirected.

Table 3.2: *Summary of previous works.*

Approaches	**Modifies/Target VMM**	**Limitation(s)**
Partial-Boost[114]	host kernel/Xen	Target single-core VM, unfair in SMP
vSlicer[174]	host kernel/Xen	Introduces extra context switches
vBalancer[71]	host kernel&guest software/Xen	Only address the IO device interrupt
AQL_sched[162]	host kernel/Xen	vCPU quantum is fixed after profiling
xBalloon[160]	host kernel&guest kernel/Xen	Still preempting vCPU during IO
vMigrater[107]	guest software/KVM	Only works for large vCPU size of VM

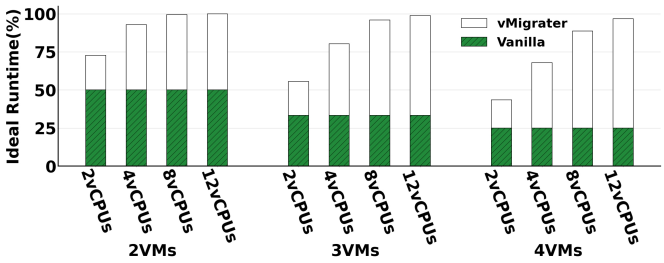


Figure 3.4: *vMigrater IO process maximum extendable runtime, 2/4/8/12 vCPUs VMs, oversubscribe ratio 2/3/4:1.*

3.3.2 Partial Boosting

Partial boosting [114] is a technique tailored for single-CPU VM scenarios within the Xen VMM framework. The authors of partial boosting have implemented a gray-box approach to identify IO-intensive processes operating on a vCPU. They propose that IO-intensive processes typically utilize minimal CPU resources and often preempt other tasks, suggesting that frequent context switches could serve as an indicator of an IO-intensive process. To monitor these context switches, the authors utilize introspection of the x86's `cr3` register, which holds a pointer to a process's page table. When IO activity is detected on a vCPU, its priority is temporarily elevated by placing it at the top of the scheduler queue. Once the vCPU completes its timeslice, it is relocated to the end of the queue to maintain fairness in scheduling. However, we contend that this method of monitoring through the `cr3` register is not sufficiently precise. Access to the `cr3` register is only available following a VM exit, meaning when the VM is not actively running. This granularity is too coarse to accurately pinpoint which specific process is driving the IO activity.

3.3.3 Task-aware Boosting

A similar idea to partial boosting is to accurately detect and boost only the process that is doing the IO. xBalloon [160] is developed to effectively boost the IO process in situations where a vCPU is handling both an IO process and a CPU-bound workload. xBalloon introduces a mechanism known as the "balloon period," during which it temporarily suspends the CPU-bound workload while the vCPU is engaged with the IO process. This adjustment allows the vCPU to focus exclusively on the IO workload temporarily. As outlined in **§ 3.2.4**, this strategy enables such a vCPU to preempt other vCPUs upon receiving an IO event, thereby reducing the latency from the moment the interrupt is received to when it is serviced. Following the completion of the IO process, xBalloon permits the CPU-bound workload to resume.

However, since the CPU-bound workload resumes immediately after the IO process and consumes an entire scheduling timeslice, a fair VM scheduler would subsequently allocate a timeslice to another VM. As a result, the IO process might have to wait for another timeslice before it can receive any subsequent interrupts. Consequently, while xBalloon enhances the responsiveness to IO events, it does not fully optimize IO performance due to the potential waiting time imposed by the fair scheduling of other VMs.

The latest work is vMigrater [107]. vMigrater keeps migrating IO thread(s) be-

tween the active vCPUs. However, it is possible that most of the vCPUs of the VM are descheduled (or inactive) at the same time. **Fig. 3.4** shows the ideal runtime of a thread that can be achieved by leveraging vMigrater idea. We utilize the `perf` tool to monitor the context switches of each vCPU and analyze the potential extension of a thread's runtime by migrating it among active vCPUs until there are no available vCPUs for further migration. In scenarios with oversubscription ratios of 2:1, 3:1, and 4:1, the native runtime for each vCPU is approximately 50%, 33%, and 25% of the total timeslice, respectively. Using vMigrater for a VM equipped with 12 vCPUs, we observed that a thread's ideal runtime could reach as high as 99.9%, 99%, and 97%. Conversely, for a VM with only 2 vCPUs, the thread's ideal runtime extends up to only 74%, 55%, and 27%. This discrepancy highlights that the performance improvement offered by vMigrater is significantly reduced for VMs with fewer vCPUs, mainly because there are fewer active vCPUs available to host the thread.

Additionally, multi-thread applications face challenges in scenarios where only one vCPU is active. In such cases, vMigrater might migrate multiple IO threads to this single active vCPU, which could actually worsen IO performance rather than enhancing it. vMigrater also encounters issues with delayed interrupt delivery. If the vCPU designated as the interrupt handler is inactive, the rescheduling IPI cannot be delivered promptly. Furthermore, the IO threads might have already migrated to an active vCPU, complicating the situation. Lastly, the implementation of vMigrater's migration service in the guest user-space poses another challenge. This aspect makes vMigrater difficult to adopt in production environments because cloud providers find it challenging to convince clients to run additional services within their VMs, especially in an oversubscribed environment.

3.4 Design

Anubis is based on the following ***design principles***:

- No modifications to the guest VM software – i.e., support legacy software stacks. Our solution should only modify the hypervisor to ease and maximize industry adoption.
- Exploit the knowledge on the current scheduling state of each vCPU.
- Maintain the priority of the vCPU executing IO, and accurately deprioritize it once IO ceases.
- Guarantee fairness between the boosted VM and other VMs.

Overarching Design. *Anubis* extends modern hypervisors fair schedulers in order to:

- **Improve the VM software responsiveness to IO events** *by boosting the scheduling priority of a vCPU* that receives an interrupt, or a rescheduling IPI; we exploit the hypervisor's scheduler knowledge to decide if to boost the priority or reroute an interrupt or IPI;
- **Accurately maximize the VM's IO performance** *by boosting a vCPU only during the IO events period*, by lightweight monitoring and introspecting each VM, in order to identify at any time if the workload on a vCPU is IO- or CPU- intensive. we introduce a heuristic to determine if an IO-intensive epoch is ongoing or ceased;

- **Maintain scheduling decision fairness** *by migrating time quantas among vCPUs of the same VM* – hence, introducing a new interpretation of scheduling fairness; we present an algorithm to conserve fairness and limit for how long vCPUs get priority boosted.

Anubis is designed around type-2 virtualization. However, we believe *Anubis* can be applied to type-1 virtualization as well.

3.4.1 Improve the VM Responsiveness

As mentioned before, in a resource oversubscribed scenario, the vCPU inactivity time reduces the vCPU responsiveness. Latency-sensitive IO applications significantly suffer as they are unable to respond promptly to the arrival of IO events. Unlike previous efforts such as those by vSlicer [174] which aimed to shorten the vCPU inactivity period, *Anubis* enhances the VM scheduler to improve vCPU responsiveness by transparently monitoring VM behavior to guide scheduling decisions. This ensures that any incoming interrupt is delivered to a vCPU with minimal delay, a crucial factor for maintaining the responsiveness of software operating within a vCPU. *Anubis* achieves this through *interrupt-boosting*. With this technique, if the destination vCPU of an interrupt is active, the hypervisor directly injects the interrupt. If the vCPU is inactive, the hypervisor reroutes the interrupt to another running vCPU within the same VM, preferably the one that was scheduled most recently. If no vCPUs in the VM are active, *Anubis* temporarily elevates the scheduling priority of the target vCPU, injects the interrupt, and wakes the vCPU, thus preempting other vCPUs. Furthermore, when the application waiting for an IO event and the interrupt processing occur on different vCPUs, a rescheduling IPI is generated to activate the IO task. *Anubis* identifies these ”interrupt-related” rescheduling IPIs, and if the target vCPU is inactive, it temporarily raises its scheduling priority, injects the IPI, and immediately wakes the vCPU.

Anubis does not focus on Single-Root IO Virtualization (SR-IOV), which benefits from IO pass-through. Tracing IO events with SR-IOV poses challenges that could introduce undesirable overheads, although potential solutions could be explored to address these issues.

3.4.2 Maximize the VM IO Performance

Unlike burstable VMs[41, 73, 133], *Anubis* specifically enhances the vCPU only during IO event periods. While this approach might seem similar to previous initiatives, such as the one implemented by [160] which requires modifications to the guest software, *Anubis* operates distinctly at the hypervisor level and does not necessitate any alterations to the guest software. *Anubis*’s strategy, termed ”accurate-boost,” aims to prevent the vCPU from being preempted during IO processing, thereby optimizing IO performance by extending the active runtime of a vCPU. Essentially, *Anubis* dynamically adjusts the CFS timeslice of vCPUs in response to IO events. Consequently, for *Anubis* to effectively boost a vCPU solely during IO event periods, it must precisely pinpoint the start and end of each vCPU’s IO event period. To achieve this, we introduce the concept of *IO Points*—a mechanism designed to dynamically determine the likelihood that a vCPU is engaged in processing IO events. This allows *Anubis* to ap-

ply boosts judiciously, ensuring enhanced performance without unnecessary resource consumption.

3.4.2.1 Accurate Detection of IO Events.

We observed that a vCPU is likely engaged in IO processing whenever it receives interrupts from IO devices or rescheduling IPIs. Moreover, certain VM exits, particularly those involving read/write operations to IO device memory, also signal IO activity. Accordingly, a vCPU will be awarded 1 *IO Point* each time it handles an IRQ, receives an IPI, or exits the VM for Memory-Mapped I/O (MMIO) reasons. During each scheduling tick, the fair scheduler evaluates whether the *IO Points* have increased since the last tick. Successfully accruing an *IO Point* within a scheduling interval suggests that the vCPU is actively involved in processing IO. This mechanism helps in dynamically assessing and responding to the IO demands of a vCPU, ensuring that the system's response is both timely and appropriate based on the vCPU's current activities.

3.4.2.2 Accurate Detection of IO Events Boundaries.

A positive *IO Points* balance indicates that a vCPU is or has been engaged in processing IO events. To fine-tune the boost and enhance system fairness, it's crucial to determine when IO events end so that boosting can be stopped earlier. Our approach draws from how the vanilla CFS manages IO tasks in non-virtualized environments. Typically, an IO task relinquishes CPU resources while waiting for IO events, allowing CFS to temporarily increase its priority for prompt preemption when the IO resumes.

Detecting the end of IO event periods within a vCPU is key to enabling it to yield computing resources right after an IO event concludes. We achieve this by monitoring the currently executing task inside a vCPU through introspection of the VM guest memory from the hypervisor. If we observe a task change from the initially detected one, it likely signifies the end of the IO activity. This insight allows us to prematurely pause a vCPU's operation, thereby restoring fairness by adjusting resource allocation more equitably among vCPUs.

3.4.2.3 Accurate Boosting Preservation.

The actual behavior of an IO task is complicated, and the boundaries of the IO and CPU-bound workloads are not straightforward. Here, we introduce 2 terms; ***IO vCPU***: the vCPU that has IO event, and *Anubis* should prioritize it by preventing it from being preempted. ***Non-IO vCPU***: the vCPU that doesn't have IO event, and *Anubis* should deprioritize it, forcing it to voluntarily yield the computing resources if in debt. In our observation, the IO vCPU might stall the IO activity for a short time, as well as a non-IO vCPU might perform IO activity for a short time. This interference can largely hurt the performance improvement of Anubis, because it might incorrectly stop boosting an IO vCPU or start boosting a non-IO vCPU.

Anubis relies on *Confidence Points* to counterbalance the impact of unstable vCPU IO activity. *Confidence Points* record the historical IO event of vCPU, thereby helping *Anubis* in making accurate recognition of the IO and non-IO vCPU. The vCPU will accumulate 1 *Confidence Point* when *IO Point* has increment in the previous scheduling

tick. In other words, *Confidence Points* indicate the vCPU had IO events during a previous scheduling tick interval.

If an IO vCPU has an ambiguous indication, such as no increment in *IO Points* but the running task still matches the one detected during the IO event. *Anubis* will decrement the current *Confidence Points* according to the *degradation policy*. Different *degradation policy* of *Confidence Points* adjust the weight of the IO event that was completed long ago. If the remaining *Confidence Points* exceed a certain threshold, *Anubis* will still classify it as an IO vCPU, disregarding the ambiguous evidence. On the other hand, the non-IO vCPU must continuously accumulate *Confidence Points* to exceed the threshold to be classified as an IO vCPU. This implies that *Anubis* will only recognize a vCPU as an IO vCPU when it is capable of upholding a steady IO event period.

3.4.3 Overall Fairness

Despite *Anubis*'s ability to accurately preserving the high priority of an IO vCPU, *Anubis* could severely deteriorate the performance of other VMs in the system due to the possible high resource demands of the IO vCPU. Mainly because of: **1.** *Anubis* allows IO vCPU to promptly preempt the running task whenever an interrupt is delivered. **2.** *Anubis* prevents IO vCPU from being preempted, which results in an extension of the IO vCPU runtime.

More computing resources need to be consumed to boost the IO vCPU, but the computing resource is limited under the oversubscribed scenario. This brought us to a trade-off: *If we want to maintain the better performance of the IO vCPU, we have to accept this unfairness.* Inspired by the policy in bare-metal scheduling, where IO and computing-intensive workloads are running together without the semantic gap of the hypervisor. The CFS prioritizes the IO workload, allowing it to preempt computing-intensive workloads in a timely manner and ensuring that it is never preempted by such workloads. As a solution to this trade-off, *Anubis* prioritizes the IO vCPU while reducing priority for the non-IO vCPU that from the same VM, thereby striving to uphold *overall fairness*.

We introduce *Anubis*'s *overall fairness* design: rather than maintaining the fairness between the boosted vCPU and the other vCPUs, *Anubis* ensures the fairness among boosted VM and the other VMs. We introduce *Anubis* debts to each VM, which is shared amongst all the vCPUs associated with the VM. Whenever a VM's vCPU extends its runtime or forcibly preempts background vCPU, the vCPU adds time to the debts. In other words, *Anubis* boosts an IO vCPU by "borrowing" the background vCPU's time. Fairness is maintained when boosted VM pays the debts back. There are two ways to maintain fairness:

Short-term fairness: Non-IO vCPU(s) pay the debts by voluntarily yielding the computing resource to the background vCPU while the IO vCPU is boosting.

Long-term fairness: All vCPUs of the VM are IO vCPUs, the debt will keep accumulating, and none of the IO vCPUs will pay the debt during the IO event until there is a non-IO vCPU.

To prevent a VM from never paying its debt, *Anubis* configures a debt threshold. If the accumulated debt surpasses the threshold, the boost will stop. It is worth mentioning that the IO vCPU doesn't need to pay the debt as long as it is identified as IO vCPU,

this can maintain the IO performance no worse than the vanilla case. The *Anubis* debt maximum limitation is configurable to provide flexibility to the Cloud provider.

3.4.4 Malicious IO Events and Applicability

If a malicious user deliberately triggers a high volume of IO events, including rescheduling IPIs, to monopolize system resources, *Anubis* incorporates configurable limits, such as a maximum debt threshold, to prevent the vCPUs of the impacted VMs from being excessively boosted—this holds until the accumulated debt is cleared.

Anubis operates optimally when each physical CPU (pCPU) is responsible for boosting just one IO vCPU. If multiple IO vCPUs are active, *Anubis* identifies this conflict, as these vCPUs may start preempting each other aggressively. In such cases, *Anubis* will attempt to redistribute the vCPUs across available pCPUs to alleviate the conflict. However, if redistribution proves ineffective, for instance, when all pCPUs in an oversubscribed system are already managing one IO vCPU, *Anubis* will select one IO vCPU to prioritize for boosting, based on predefined configurable policies.

3.4.5 Summary of Anubis' Features

Responsiveness. *Anubis* always ensures the responsiveness of the vCPU when an interrupt arrives. *Anubis* identifies IO activity in a vCPU using *IO Points*. When a vCPU receives an interrupt, *Anubis* ensures the vCPU can forcibly preempt the running task. The unfinished timeslice of background vCPU will accumulate to the boosted VM's debt.

Boosting. *Anubis* prevents IO vCPU from being preempted during the IO event period. *Anubis* extends the runtime of IO vCPU, and the extended time will accumulate to the debt. The IO vCPU can be preempted only if it doesn't gain any *IO Points* and *Confidence Points* is lower than the threshold, or its runtime has exceeded the maximum timeslice of the scheduler.

Accurate. *Anubis* accurately detects the boundaries of IO events inside vCPU. *Anubis* forces non-IO vCPU to yield to pay the debts. It is important to mention that, if a non-IO vCPU is doing the IO event, while its *Confidence Points* has not yet exceeded the threshold, it won't be required to pay the debts. However, the non-IO vCPU will also not be recognized as IO vCPU, so it will not be able to extend its runtime.

Fairness. *Anubis* provides overall fairness. *Anubis* checks the debt of the VM at each scheduling tick. If a VM is in debt and non-IO vCPU(s), the VM will pay the debts. *Anubis* only lets the non-IO vCPU that is not doing IO event yield computing resources to pay the debt. The debts of VM will be reduced by the time of the unfinished part of the timeslice that the yielding vCPU was supposed to run.

3.5 Implementation

We developed a prototype of *Anubis* using Linux kernel version 5.10.90. Our implementation added approximately ∼2800 lines of code (LoC) to the vanilla Linux kernel and another ∼1800 LoC in various automation scripts. The primary enhancements were made to the Linux CFS and the KVM subsystem, specifically for Intel x86_64 architectures. *Anubis* is designed to schedule tasks marked with the `PF_VCPU` flag, which identifies them as vCPUs. Furthermore, *Anubis* includes a `proc` interface that allows for the selective scheduling or non-scheduling of specific VMs, offering greater control over task management. For the user-space component of the KVM hypervisor, we utilized QEMU 6.2.0. Importantly, we made no modifications to QEMU, ensuring that *Anubis* could be used with any similar software, thus enhancing its adaptability to various deployment scenarios.

3.5.1 Interrupt Redirection and Boosting

To implement interrupt-boosting, we introduced interrupt redirection and IPI boosting in Linux KVM. In Linux, each interrupt is bound to a fixed CPU or set of CPUs. For example, the disk I/O interrupt is usually bound to CPU0.

We implement interrupt redirection by using Intel's logical interrupt model and rerouting the interrupt to a different vCPU at function `kvm_arch_set_irq_inatomic`. However, after version 4.14 [112, 113], Linux only supports fixed interrupt routing, missing support for the logic interrupt model. Thus, currently, several Linux developers are working on supporting the logic delivery model on recent kernels [136]. At the same time, Intel [171] stated that the non-fixed interrupt is not supported anymore in the `x2-apic` model. Anyway, in our setup, the guest kernel uses the vanilla Linux kernel 4.14.0, and in the boot-up command we have added `-nox2apic` flag to disable the `x2-apic`. Hence, the interrupt can be redirected to any vCPU.

As mentioned in **§ 3.2.3**, it is possible that the IO process is not running on the vCPU that receives an interrupt. The vCPU will generate a rescheduling IPI and will send it to the vCPU that has the IO process. *Anubis* also checks the destination vCPU of the rescheduling IPI in function `kvm_apic_send_ipi`. If the vCPU is not running, CFS will force the running task to yield to this vCPU. If there are more than 2 tasks in the CFS `runqueue`, we will use the `set_next_buddy()` function to make sure the next wake-up task is the vCPU we are trying to boost. As a result, the IO process inside the boosted vCPU is able to wake up on time. Thus, the responsiveness of vCPU is guaranteed.

3.5.2 Accurate Boosting

Interrupt redirection and boosting are well-suited for handling short-lived IO events. However, the scenario changes when an IO event extends over a longer duration, such as during continuous disk reads. Unlike previous approaches like [114], which inferred vCPU IO activity based on the rate of context switches, *Anubis* employs a series of strategies specifically designed to accurately determine when a vCPU is engaged in IO tasks. The details of these strategies and their implementation are outlined below.

IO Points. The occurrence of the delivery of an IO device interrupt or rescheduling IPI indicates a potential IO event that would happen in vCPU. In x86 virtualization, an MMIO fault is signaled by a vmx `ept_misconfig` fault, indicating that the vCPU is involved in the IO event. We implemented a per-vCPU variable called *IO Points*, which is a new feature we added in each vCPU `struct task_struct`. vCPU will gain 1 *IO Point* whenever an IO device interrupts, rescheduling IPI, or KVM exit reason with MMIO occurs. *Anubis* compares the current vCPU *IO Points* with the previous recording in each schedule tick, implemented in `check_preempt_tick()`. Any increment of the *IO Points* indicates the vCPU had a potential IO event in the previous schedule tick.

IO Task Introspection. However, previously introduced evidence doesn't reveal the actual cessation time of the IO event. Therefore, *Anubis* monitors the running task of a vCPU to detect the end of the IO event. Previous work [114] relies on the `cr3` register. `cr3` register stores the address of the running task's page table directory (`pgd`). Changing of `cr3` register indicates changing of running task. However, the guest `cr3` register is only observable during the KVM exit period. The host cannot inspect the running task of vCPU when VM stays in the guest mode. Instead, *Anubis* leverages the `current_task` pointer, similarly to earlier works [68, 69, 84]. The `gs` (per-CPU register) base address plus `current_task` address becomes the per-CPU variable `*current`, and this pointer stores the running task address. Changing of `*current` indicates changing of running task. Because the `current_task` pointer is statically compiled, the address of `current_task` can be found in the kernel symbol map(`System.map`). The Cloud providers can easily get the address of `current_task` since they are the VM kernel provider.

However, Cloud providers can not acquire the `System.map` if clients use their customized kernel. Therefore, we have implemented a heuristic approach. Because the `current_task` address is a fixed address in kernel space, we can keep adding the offset of a pointer length to the `gs` base address and read the value until the boundary, the next vCPU's `gs` base address. We will have few candidates, and most of them can be eliminated in 3 steps: **1.** `current_task` is a per-CPU variable, the offset to the `gs` base remains the same for each vCPU; **2.** `current_task` value changes over time; **3.** `current_task` value is a kernel space address. Initial testing shown this method working also with Kernel Address Space Layout Randomization (KASLR) or Function Granular KASLR (FG-KASLR)[128]. While *Anubis* mainly targets Linux guest VMs, we expect *Anubis* working also with other traditional OSes. For example, in FreeBSD, we could start introspecting `curproc` instead of `current_task` in Linux.

Confidence Points. Guest `*current` address can be mapped to host userspace via the QEMU `memory_slice`. Implemented in `handle_ept_misconfig`, the host reads the `*current` value during KVM exit reason for MMIO and marks the value as *IO possible*. In each schedule tick the host reads the `*current` again and compares the value with *IO possible*. A mismatch indicates the end of the IO event. However, the mistake is unavoidable if more than one thread is related to the IO event because each time the host only reads one value out. *Confidence Points* can help us to solve

this issue.

Like *IO Point*, the *Confidence Point* is a per vCPU variable, which is a new feature we have implemented in the vCPU *struct task_struct*. In each schedule tick, vCPU can get 1 *Confidence Point* if vCPU has gained *IO Points* in the previous schedule tick, implemented in `check_preempt_tick()`. The higher the *Confidence Points*, the more IO activities have been done previously for this vCPU.

In contrast, the non-IO vCPU has to continue gaining *Confidence Points* until it is larger than the threshold to be considered an IO vCPU, as an IO vCPU is supposed to maintain a steady IO event.

The current degradation policy of *Anubis* in using is to divide *Confidence Points* by two. For example, with a threshold set as four, if an IO vCPU has a steady IO event in the previous 100 schedule ticks. After the IO event is completed, this IO vCPU will lose all *Confidence Points* in five ticks and be recognized as a non-IO vCPU. On the contrary, a non-IO vCPU needs a steady IO event lasting at least four schedule ticks to be considered as an IO vCPU. Higher *Confidence Points* threshold indicates it is harder to recognize a vCPU as an IO vCPU.

Confidence Points degradation policy and the *Confidence Points* threshold are fully configurable.

3.5.3 *Anubis* Debt System

Anubis maintains fairness among VMs. We implemented a feature in each `struct kvm` structure called debt. Unlike the current burst VM used in the Cloud [41, 73, 133], the *Anubis* debt gives more granularity by boosting vCPU solely during the IO event period, rather than being constantly active. Each vCPU of the VM will contribute to the debts by accumulating the time gained from other background vCPUs, and vCPUs of the same VM can synchronize access to this shared variable, controlled by `spin-lock`. Whenever a vCPU preempts another vCPU or extends its runtime, the vCPU will accumulate the "borrowed" time to its debts.

Anubis checks the debts at every scheduled tick, implemented in `check_preempt_tick()`. *Anubis* set a *Maximum Debt* threshold to prevent the IO vCPU boosting endlessly. If the total debt reaches such threshold, the IO vCPU boost will be disabled; it can't preempt other vCPU when an interrupt is delivered or maximize the IO performance by extending its runtime when it gains *IO Points*. The*Maximum Debt* threshold is fully dynamically configurable; for example, a cloud provider can set the threshold based on its cost structure.

3.5.4 *Anubis* Policies

Anubis features configurable policies to handle scenarios where one IO vCPU is preempted by another, a situation evident when both have *Confidence Points* exceeding a set threshold, indicating the presence of multiple IO vCPUs on the same pCPU. Initially, *Anubis* attempts to resolve this by rearranging the vCPUs, swapping the core affinity of an IO vCPU with a non-IO vCPU from the same VM. If this rearrangement proves ineffective—such as in cases where all vCPUs within the VM are designated as IO vCPUs—*Anubis* then resorts to an alternative policy. By default, *Anubis* boosts the IO vCPU that has accumulated the highest *Confidence Points*. Nevertheless, this

decision is adjustable; alternatives include boosting the IO vCPU with the lowest debt or choosing not to boost any vCPU at all.

In essence, *Anubis* is designed to manage the aggressive preemption triggered by interrupt delivery among multiple IO vCPUs effectively. Although this can enhance performance, in an oversubscribed environment with constrained computing resources, *Anubis* strategically allows only one IO vCPU to maximize its IO performance by extending its runtime.

3.6 Evaluation

Testbed. We conducted experiments on a custom-built server equipped with a Supermicro X11DPi-N motherboard, featuring 768GB of DRAM and dual Intel(R) Xeon(R) Gold 6230R CPUs, each with 26 cores operating at 2.1GHz. To ensure more consistent results, hyperthreading was disabled, even though it is supported by Anubis. Similarly, for stability and repeatability in performance measurements, all experiments were conducted with the CPU performance profile, which includes disabled frequency scaling and turbo boost. The VMs used in the tests were configured with 8GB of RAM and varied between 2 and 8 vCPUs. Each guest was running Ubuntu 18.04 LTS with an unmodified Linux kernel version 4.14.0-041400-generic. The host system was running Ubuntu 22.04 LTS with a Linux kernel version 5.10.90, which had been extended with Anubis. The host kernel's scheduling frequency was set to $1kHz$, meaning the scheduling tick was $1ms$, and the scheduler period (`sched_latency_ns`) was set to $8ms$, following Red Hat's recommendation [97].

For the experiments, we examined VM oversubscription ratios of 2:1, 3:1, and 4:1 vCPUs to pCPU, which we determined to be the most relevant for current industry practices, based on various sources on elastic Cloud specifications [40, 64, 74, 76, 93, 94, 95, 103, 123, 142, 168]. However, it is important to note that *Anubis* imposes no restrictions on the actual oversubscription ratios used. In our experiments, all the pCPUs reside in the same NUMA node, and we divided those pCPUs into two pools [141]. A computing pool, depending on the vCPUs size of the VM we test, 2 to 8 pCPUs, which hosts all vCPUs – with a 2:1, 3:1, or 4:1 oversubscribed ratio; and another pool, 2 to 8 pCPUs hosting anything else, including all vHost and QEMU-IO threads. This ensures that only vCPUs contend for the same pCPU resource.

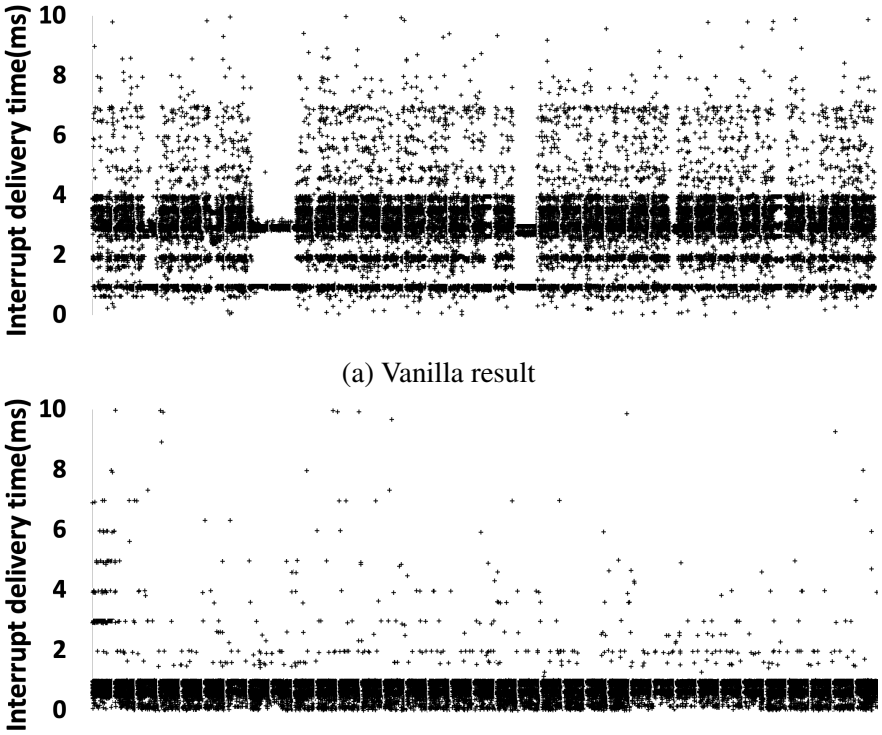
Table 3.3: *Macro-benchmarks test set*

Application	**Description**
HBase	Sequential scan records with YCSB [176].
HDFS	Sequential read 26GB with TestDFSIO [96].
MySQL	`seqrd` and `seqwr` 10GB files with sysbench [161].
MongoDB	Insert with MongoDB-performance-test [145].
Nginx	Concurrent requests with ApacheBench [45].
Postmark	Simulate mail servers files operations [146].
Redis	Set values with Redis-benchmark [150].

Benchmarks. **Table 3.3** lists the real-world workloads we used in the evaluation. We run the same compute-intensive workload, sysbench-cpu benchmark [161], for all

the VMs to create background pressure, and one VM runs the IO-intensive workload as the test case. Our upper bound is the non-oversubscribe case: each vCPU of the VM is running solely on a pCPU.

Experiments. We evaluated how **§ 3.6.1**: *Anubis* shortens the vCPU inactivity time; **§ 3.6.2**: *Anubis* improves the VM's IO performance under the oversubscribe scenario; **§ 3.6.3**: *Anubis* maintains short-term and long-term fairness; but also **§ 3.6.4**: *Anubis* overheads; **§ 3.6.5**: *Anubis* vs latest work; and **§ 3.6.6**: *Anubis* in serverless computing scenarios.



(a) Vanilla result

(b) *Anubis* result

Figure 3.5: *Time interval between KVM delivered interrupt and receiver vCPU is active, 4 vCPUs VMs, oversubscribe ratio 2:1*

3.6.1 vCPU Responsiveness

Boosting vCPU responsiveness through device interrupt, rescheduling IPI, and interrupt redirecting is a cornerstone of our approach with *Anubis*. To gauge how *Anubis* reduces interrupt pending times, we employed Linux kernel event tracepoints. For each incoming interrupt (whether it's a device interrupt or a rescheduling IPI), we first capture the moment the KVM delivers the interrupt to a vCPU, marking this with a timestamp at the function `__apic_accept_irq`. Subsequently, we log the moment the recipient vCPU becomes active, which is traced at the scheduler event tracepoint `sched_switch`. The duration between these two events is then measured. If the vCPU receiving the interrupt is active at the time the interrupt is delivered, the time

interval recorded would be zero, indicating no delay in handling the interrupt due to the vCPU already being active.

Our experiments involved 2 VMs, each with 4 vCPUs, operating under a 4:1 oversubscription ratio. We utilized an Nginx server for IO-intensive tasks and ran sysbench-cpu for compute-intensive demands across both VMs. Throughout these experiments, we tracked around 35,000 interrupts to assess the enhancements *Anubis* brings to system responsiveness. The experimental results are presented in **Fig. 3.5**. The y-axis represents the time interval expressed in milliseconds, and the x-axis denotes each interrupt measurement.

The vanilla case results are depicted in **Fig. 3.5a**. Here, we observe that the average interval time between the delivery of the interrupt and the vCPU resuming operation is approximately 3-4 ms. In the worst-case scenario, expected as we analyzed in **§ 3.2.4**, the interrupt comes right after the vCPU got preempted. This duration can extend up to about 7-8 ms, equivalent to roughly 1 schedule timeslice of a task entity on CFS in our setup. Additionally, if the IO thread is not executing on this vCPU, an extra rescheduling IPI is necessitated. In the worst-case scenario, a maximum of 14-16 ms(twice the maximum delay) must pass before the VM can respond to the interrupt. Notice that we only trace the time interval of the individual interrupt; the result in Fig doesn't show the accumulated delay time of the consecutive IO device interrupts + rescheduling IPI delivery mentioned above.

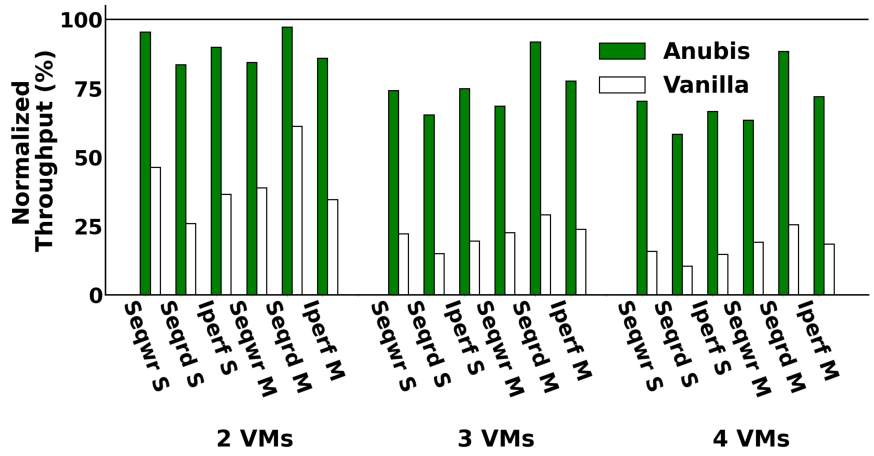
On the contrary, **Fig. 3.5b** showcases results from *Anubis*, which utilizes interrupt redirecting and boosting. Here, the vCPU typically resumes operations within 1 ms of receiving an interrupt, significantly reducing the response time. While this setup does not ensure that the active vCPU processes the interrupt instantly after reception, the delay is markedly decreased, with average reductions around 350% per interrupt, demonstrating the effectiveness of *Anubis* in optimizing interrupt handling in virtualized environments.

3.6.2 IO Performance

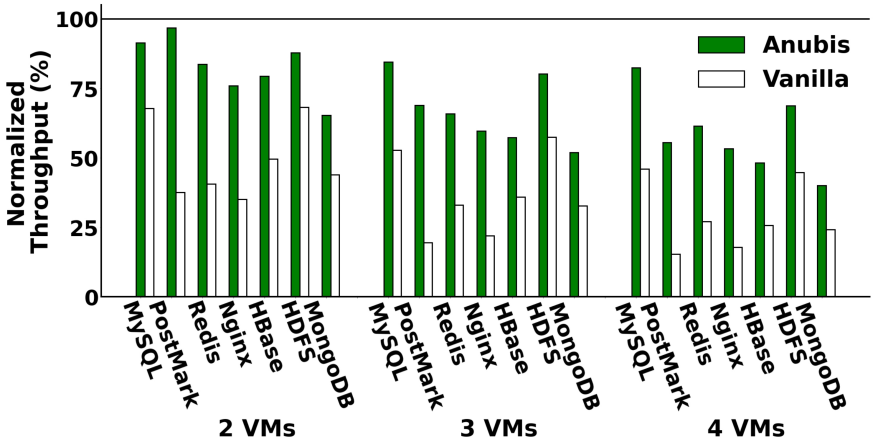
Here we tested *Anubis*'s performance improvement compared with the vanilla case for both disk IO and network IO applications. Our upper bound is the non-oversubscribe case, in which the vCPU is solely running on a pCPU. Results show in **Fig. 3.6**.

Micro-benchmark. We conducted a series of disk read and write tests using sysbench-fileio[161], and iPerf3[105] for network IO testing. The results are displayed in **Fig. 3.6a**, and normalized to the non-oversubscribe case. With *Anubis* enabled, on average, there is a notable performance improvement when compared to the vanilla case, ranging from 59% to 500%, and the performance has improved from 45% to 97% compared to non-oversubscribe VM.

Macro-benchmark. We tested both single-thread and multi-thread real-world applications, including HDFS[99], LEMP[117], MongoDB[134], Postmark[146], Redis-server[**?**], HBase[98], Hadoop[96], and MySQL[161]. The macro application setup details and its corresponding benchmark we chose are listed in **Table 3.3**. Results are normalized to the non-oversubscribe case and presented in **Fig. 3.6b**. On average, with



(a) *Micro. "S" single thread, "M" multi-threads)*



(b) *Macro*

Figure 3.6: *Benchmarks results, normalized to the non-oversubscribe case, 4 vCPUs VMs, oversubscribed ratio of 2/3/4:1*

Anubis, the throughputs of these benchmarks are improved by about 70%, 84%, and 103% compared to vanilla KVM for the settings with oversubscribed ratios of 2:1, 3:1, and 4:1 respectively. In terms of the non-oversubscribe case, the *Anubis* improves the performance up to 40% to 97% of the non-oversubscribe VM performance.

Both micro and macro benchmarks show that with a higher oversubscribed ratio, on average the degree of improvement relative to the vanilla case remains approximately constant. However, achieving the performance of non-oversubscribed cases becomes increasingly difficult. This can be attributed to *Anubis*'s design, which takes care of fairness among the VMs.

Different number of vCPUs. We run VMs with different numbers of vCPUs to demonstrate that *Anubis* can still meet the promised performance improvements. The experiments run the same macro benchmark set, with an oversubscribed ratio of 2:1, and each VM with 2, 4, and 8 vCPU, the results show in **Fig. 3.7**. We can observe that the different numbers of vCPUs do not affect the performance improvement of Anubis. Additionally, the performance improvement of *Anubis* can be even better if

the VM has many vCPUs. Because the more non-IO vCPUs, the easier the fairness to achieve, *Anubis* can boost the IO vCPU more aggressively. On average, the performance improvement remains about 38% to 500% compared with the Vanilla case, and 60% to 97% compared with the upper bound non-oversubscribe case.

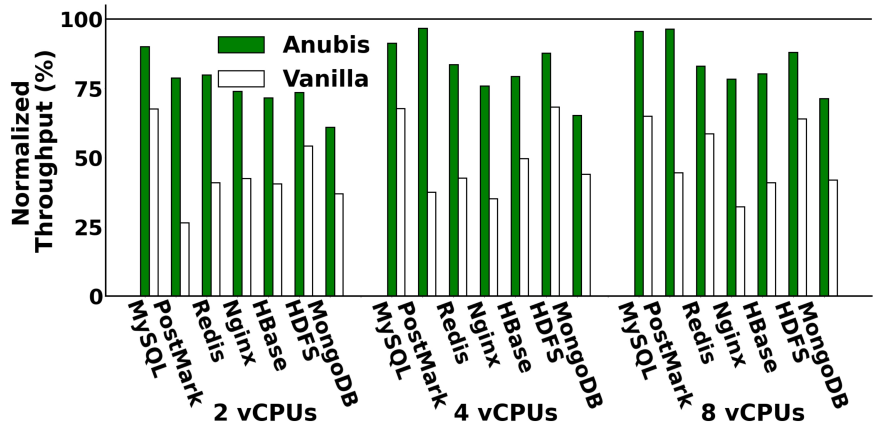


Figure 3.7: *Macro-benchmark, normalized to non-oversubscribe case, 2/4/8 vCPUs, oversubscribed ratio 2:1*

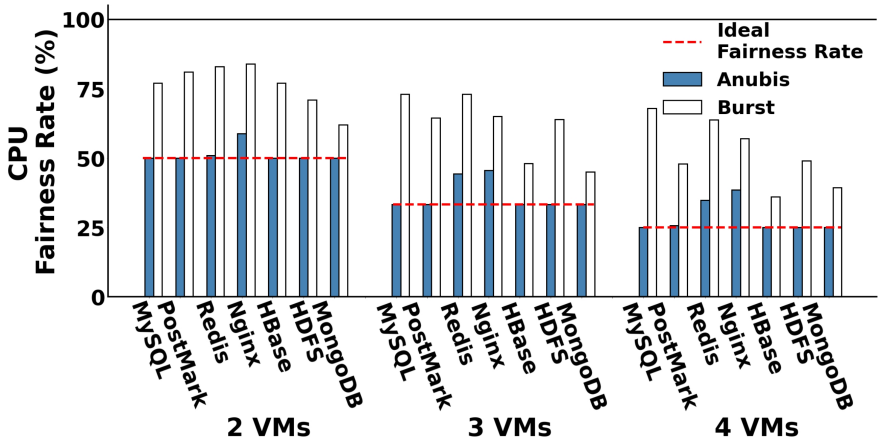
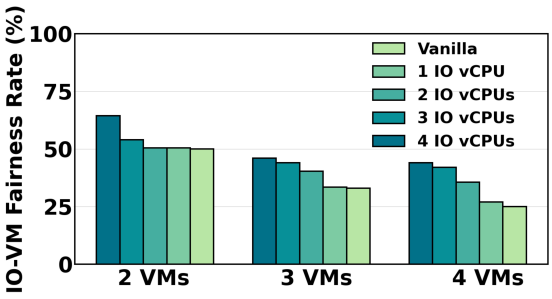


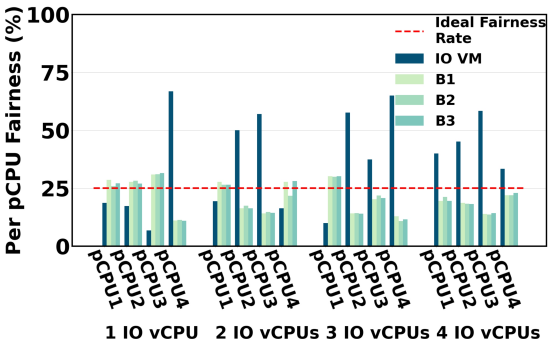
Figure 3.8: *Fairness of Burst VM to achieve the Anubis performance, 4 vCPUs, oversubscribed ratio 2/3/4:1*

3.6.3 Overall Fairness

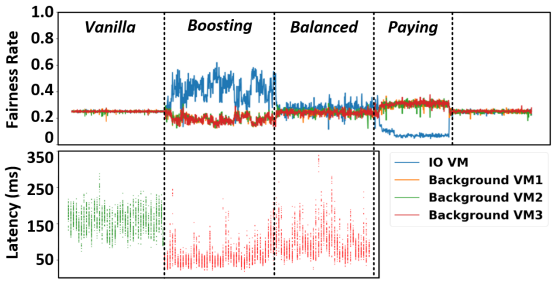
This section illustrates how *Anubis* guarantees fairness among oversubscribed VMs. **§ 3.6.3.1** shows that *Anubis* is not a simple boost like burst VM. **§ 3.6.3.2** illustrates how *Anubis* ensures short-term fairness **§ 3.6.3.3** describes how *Anubis* guarantees long-term fairness. In the following experiments, we use `perf` to trace the context switch for each vCPU task, and based on that, we can correctly accumulate the actual execution time of each vCPU task.



(a) *Short-term fairness, different number of IO vCPU*



(b) *Short-term fairness, per pCPU in detail*



(c) *Long-term fairness and performance in different stages*

Figure 3.9: *Anubis fairness evaluation*

3.6.3.1 *Anubis* vs Burst.

To substantiate the delta of our approach versus a naive boost solution, we set up an experiment to illustrate how *Anubis* is different compared to the burst VM in Cloud [41, 73, 133]. **Fig. 3.8** shows the fairness results of the 4 vCPU VM under-oversubscribe and running the same macro benchmark sets. The performance results are the same as **Fig. 3.6b**. The most interesting part is the white bar, the *Burst*, which indicates how many CPU resources the oversubscribed vanilla VM needs to burst to achieve the same performance as Anubis. With the same performance improvement, *Anubis* offers a nearly ideal fairness rate, which proves that *Anubis* can accurately detect and boost the IO vCPU.

3.6.3.2 Ensure Short-term Fairness.

As mentioned in **§ 3.4.3**, non-IO vCPUs can yield computing resources(paying debts) while the IO vCPU is boosting(borrowing debts). The more non-IO vCPUs, the better the fairness can be achieved. Thus, we have picked single-thread and multi-thread IO-intensive workloads to show the difference. We use a 4 vCPUs VM, and we show the fairness between the boosted VM and the background VMs with oversubscribed ratio from 2:1 to 4:1.

Fig. 3.9a presents the overall fairness of the boosted VM during the Boosting time. The 4 cases are as follows: *1-IO-vCPU:* The sysbench-seqrd thread is pinned to the vCPU that handles the interrupt. *2-IO-vCPU:* The redis-server thread is pinned to the vCPU that doesn't handle the IO device interrupt, while the vCPU handling the interrupt is also marked as an IO vCPU. *3-IO-vCPU and 4-IO-vCPU:* Each PHP worker and the Nginx server are pinned to a vCPU, while *3-IO-vCPU* in this case we leave 1 vCPU as a non-IO vCPU.

In the scenario of *1-IO-vCPU*, the fairness across VMs is closely related to the base case due to a higher number of non-IO vCPUs. However, as the number of the IO vCPU increases, short-term fairness begins to deteriorate due to the scarcity of the non-IO vCPU available to offset the debts accumulated during the Boosting stage. Detailed insight into the achievement of short-term fairness is depicted in **Fig. 3.9b**. Which presents CPU resource usage per vCPU in the case of an oversubscribed ratio of 4:1. Observations reveal that *Anubis* facilitates non-IO vCPUs in yielding CPU resources to background vCPUs, while correctly boosting IO vCPUs.

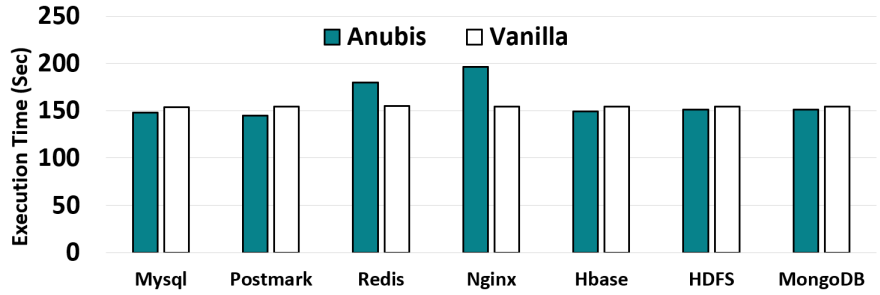
Despite these insights, it's worth noting that short-term fairness can be better. In our experimental setup, we created a worst-case distributed scenario by forcibly assigning the IO threads to different vCPUs to showcase the worst-case fairness of Anubis. In the following section, we will discuss how *Anubis* can maintain long-term fairness, especially in the context of multi-thread IO-intensive workloads.

3.6.3.3 Guarantee Long-term Fairness.

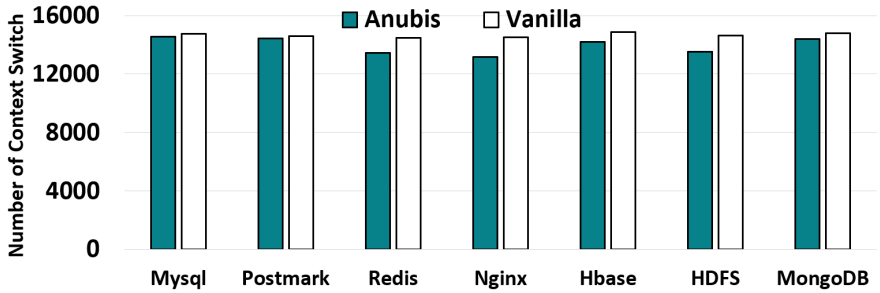
In the *Anubis* debts system, there are 3 stages during the execution time. **Boosting**: The debts are increasing, IO vCPUs are demanding more running time and keep borrowing debts, and less payback from non-IO vCPUs. **Balanced**: The debts are dynamically stable, with no increase or reduction. This could happen either by debts having reached the maximum limitation or enough payback of the debts by the non-IO vCPUs. **Paying**: The debts are reducing, and non-IO vCPUs pay more debts than IO vCPUs borrow, which usually happens when the IO event is finished. In this experiment, we use the Nginx server as the test benchmark as it is a complicated multi-thread IO-intensive workload.

Fig. 3.9c shows how the VM fairness and the Nginx performance change during the Boosting, Balanced, and Paying stages. The x-axis is sampled every 100 ms, we used `perf` to record the context switch and accumulate the actual runtime of each vCPU within the 100 ms interval to generate the fairness rate. Initially, we execute the Nginx server [137] under the vanilla case, recording per-request latency from the Apache benchmark [45]. In the middle of execution, *Anubis* is activated, resulting in a boosted state for the VM. In return, the latency is reduced during the Boosting stage. To avoid

significant unfairness, the VM's boosting is curtailed once the accumulated debt hits the maximum limitation, marking the start of the Balanced stage. Owing to the IO vCPU's exemption from paying debt, the performance declines but remains **no worse** than the vanilla case. Following the completion of all requests by the Nginx server, *Anubis* identifies the absence of any IO event. The IO vCPUs change to non-IO vCPUs and start to pay back the debt by yielding computing resources to the background VMs. The experiment proves that *Anubis* can guarantee long-term fairness.



(a) *Background computing-intensive workload execution time, measured during Boosting stage*



(b) *Number of Context Switch, measured during Boosting stage*

Figure 3.10: *Anubis overheads evaluation*

3.6.4 Overhead Assessment

We evaluated the impact of *Anubis* on the performance of background VM workloads by conducting experiments with a 4:1 oversubscription ratio, using a VM equipped with 4 vCPUs. We utilized a set of macro benchmarks, including a multi-threaded prime number calculator, to assess the performance impact on these background VMs. The execution times, represented on the y-axis, were measured during the Boosting stage of *Anubis*, as we hypothesized that any long-term effects on execution times would be negligible, as shown in **Fig. 3.9c**. Results shown in **Fig. 3.10a** generally indicate that the background VMs' performance remains unaffected by *Anubis* in most scenarios. However, in cases where the boosted IO workload involves many IO vCPUs, such as with Nginx, the performance of the background VM can drop to about 75% of that in the vanilla (unmodified) case.

To address concerns that *Anubis* might increase the frequency of costly VM context switches, similar to previous solutions like [174], we utilized the `perf` tool to tally the total number of context switches for each VM and compared these figures to those

from the vanilla setup across various oversubscription ratios. As shown in **Fig. 3.10b**, the number of context switches actually decreased in several instances. This reduction is attributed to the IO vCPUs maintaining operation due to accrued *IO Points*, thus avoiding preemption and reducing the need for context switching, particularly in scenarios with multiple IO vCPUs, like the Nginx case. In other scenarios, the number of context switches remained comparable to the vanilla case. Additionally, we measured the computational overhead of *Anubis*' introspection using the `perf` tool during each scheduling interval (every 1 ms). The introspection process incurs a cost of about 150 CPU cycles, which is relatively minor. This finding indicates that while *Anubis* is efficient, there is still potential for further optimization to reduce its computational footprint even more.

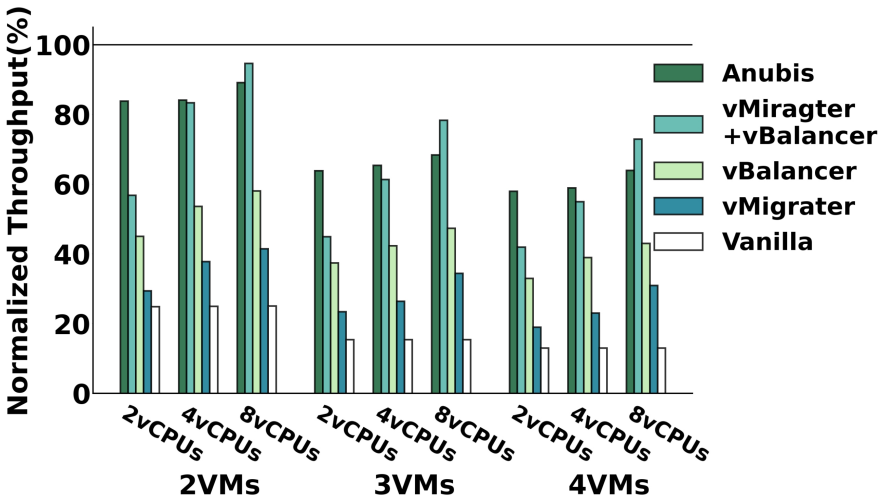


Figure 3.11: *Anubis vs Previous works*

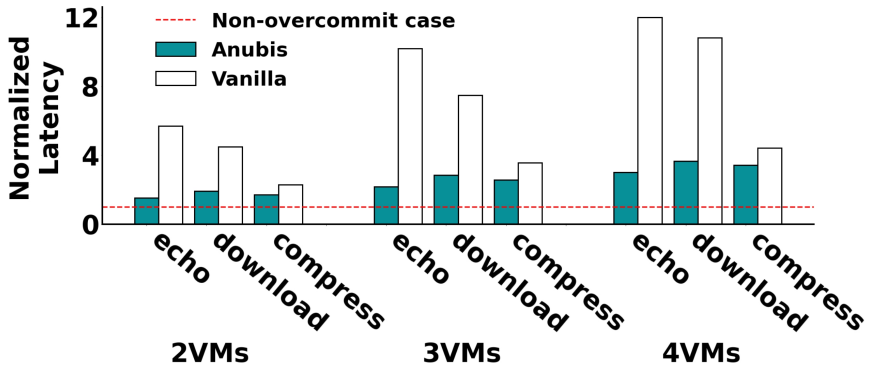


Figure 3.12: *Anubis + FaaS(OpenLambda) evaluation*

3.6.5 *Anubis* vs Previous Works

We compared *Anubis* with vanilla Linux, vBalancer [71], vMigrator [107], and the combination of vMigrator with vBalancer – this is because vMigrator must run with vBalancer to enable interrupt redirection. We primarily compared with the result of

vMigrator because, in their paper, they have compared with other approaches, vSlicer[174], xBalloon[160], and none of them have better performance compared with the vMigrater. We didn't use the open-source code from the vMigrater because it does need kernel modifications from vBalancer, unlike what we originally understood from the paper [1]. Because the interrupt redirection is part of the *Anubis* design, we can emulate the vBalancer by just enabling the IO device interrupt redirection of Anubis.

In this experiment, we ran VMs that have different vCPU sizes from 2 to 8 vCPUs, and with the oversubscribed ratio from 2:1 to 4:1, the performance results are shown in **Fig. 3.11**.

As we explained previously, if we run the vMigrater only, the performance of the IO application will **not** be increased because the vCPU that receives the IO device interrupt will not send the rescheduling IPI if it is not running. Additionally, as **Fig. 3.4** shows, under the oversubscribed ratio of 2/3/4:1, with the 2 vCPU VM, the IO thread runtime can only extend up to 74%/54%/47% of maximum timeslice. While in the 4 vCPUs and 8 vCPUs VM, the IO thread ideal extendable runtime can extend up to 90%/78%/60% and 100%/95%/88% of maximum timeslice. As we expected, the experiment result shows that the improvement of vMigrater+vBalancer to the IO application is minimal in the 2 vCPUs VM, and the vMigrater+vBalancer approach starts to work in the 4 and 8 vCPUs VM cases. However, *Anubis* offers similar results to the vMigrater approach and applies to the VMs with different numbers of vCPUs. While we tested our emulated vMigrater across all benchmarks, due to the lack of original kernel support, our results diverged from those reported in their paper – they are worse, making it difficult for us to validate the correctness of our results. Therefore, we only report the sysbench-seqrd as it was the most stable result we achieved. After comparing the results in their paper, which are also normalized to the non-oversubscribe case, we found that in some benchmarks vMigrater can achieve slightly better results than Anubis. However, vMigrater requires a VM to have many vCPUs (12 vCPUs) to accomplish this, whereas *Anubis* has no such prerequisites. Moreover, since *Anubis* doesn't necessitate modifications to the guest VM, it is more adaptable and flexible for implementation in a real Cloud environment.

3.6.6 Serverless Computing Scenario

Function as a Service (FaaS) is an emerging paradigm in which users upload and execute small, discrete pieces of code, referred to as *functions*, in the cloud. Major cloud providers have already integrated FaaS solutions into their offerings [39, 92, 132]. A study on Azure Functions shows that the execution time for serverless functions is less than 10 seconds for 90% of the functions [154]. Given this characteristic of serverless computing, cloud providers are keen to consolidate a large number of function instances onto a single physical host [35]. For our experiment in serverless computing, we utilized the open-source OpenLambda [100] FaaS framework. We set up one OpenLambda server per VM, each configured to handle up to 100 function spawns simultaneously. We registered three different functions for this experiment: echo, download, and compress services. **Echo**: In this function, the client sends data to the server,

[1] We did exchange several emails with the authors to reach such a conclusion. Moreover, we were unable to get a working code from the authors.

which then echoes the same data back to the client. **Download**: We established a VM on another host to serve as a storage machine, which is connected to the OpenLambda VM. The client sends an HTTP request containing an encoded descriptor to the server. The Lambda function decodes this descriptor, locates the target data on the storage machine, and then transfers the data back to the client through the network. **Compression**: This function simulates a video streaming service. The client submits a request to the Lambda server with an encoded video ID and a specified level of video quality. The Lambda function decodes the video ID, retrieves 100 picture frames from the video on the storage machine, compresses them to the requested quality, and sends them to another storage machine.

We have measured the latency of each request and compared the result to both vanilla and non-oversubscribe cases, and the results are in **Fig. 3.12**. We can see that the compression service gets lower improvement compared with the Echo service. Because the compression service also includes the computing-intensive workload, but *Anubis* only boosts the IO-intensive part of the service.

3.7 *Anubis* conclusion

This work identifies why low-latency IO cannot be consistently guaranteed in virtualized setups with oversubscribed resources, particularly focusing on CPU resources, an area that has not been thoroughly examined previously. Prior studies predominantly address the direct reason for slow IO, which is that the vCPU handling the IO task gets descheduled. Our research, however, highlights an indirect reason: IO device interrupts or rescheduling IPIs are not dispatched to vCPUs immediately. We introduce *Anubis*, a new IO-aware VM scheduler that precisely identifies when a vCPU is processing IO and accurately boosts the vCPU's priority only during IO processing. Notably, *Anubis* achieves this without requiring any modifications to guest software. In summary, *Anubis* enhances virtualized system performance through several key mechanisms:

1. *Anubis* mitigates the impact of vCPU inactivity by not merely shortening the inactive period of a vCPU, but by improving the responsiveness of the vCPU. It ensures that the vCPU is promptly awakened upon receiving a pending interrupt. **2.** *Anubis* maximizes the IO performance of an IO vCPU during IO event periods. Instead of simply ensuring the priority of IO tasks within the vCPU, *Anubis* enhances the entire IO vCPU's priority by maximizing its active runtime during these periods. **3.** *Anubis* maintains overall fairness among VMs while enhancing IO performance. Rather than just balancing the fairness between IO vCPUs and others, *Anubis* employs a debt system to ensure equitable resource distribution among all VMs.

This chapter includes several experimental results that demonstrate the efficacy of *Anubis* and its advantages over current state-of-the-art solutions. The code is available at `https://github.com/systems-nuts/Anubis`.

Chapter 4

Heterogeneous Fused Kernel

4.1 Introduction

Heterogeneous-ISA systems have garnered significant interest in recent years by providing a "lighter" and more programmable approach to heterogeneity than the adoption of accelerators, making it possible to run existing general-purpose code with greater efficiency than running with a single instruction set [81, 167]. However, building and running these systems presents inherent challenges due to the various fundamental mismatches between hardware and software that come with integrating different ISAs. To date, commercially available platforms focused on loosely-coupled systems without hardware cache coherence, for example, those based around PCIe. As a result, heterogeneity in ISA is still considered a liability rather than an opportunity, with its introduction leading to overheads and system-level complexity rather than an introduction of ISA-generic capabilities that provide for the greater efficiency. Instead, today's heterogeneous-ISA platforms run a separate software stack per CPU. Applications running thereon are perceived to be in a distributed system and hence cannot leverage per-platform optimizations.

New HW landscape. The hardware landscape is rapidly changing. There are several PCIe extensions that consider cache-coherent shared memory (including CXL [78], OpenCAPI [140] and CCIX [70]), making tighter interconnection of heterogeneous

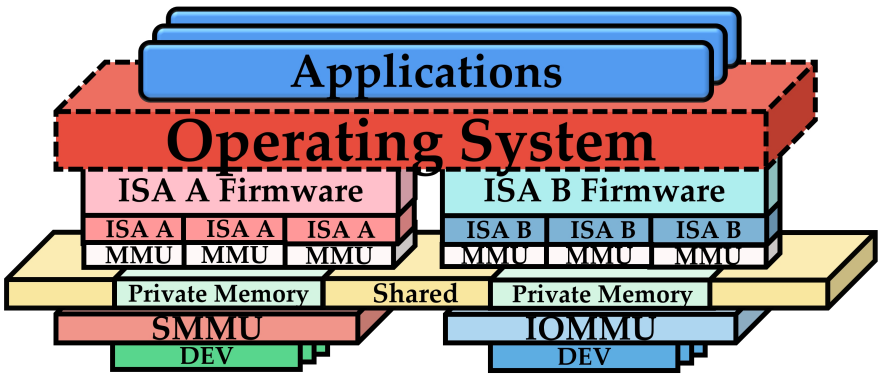


Figure 4.1: Stramash's target hardware and software model

processing units an inevitable trend. Recently, several works [116, 119, 130] have explored the concept of CXL-based remote memory pooling to effectively leverage memory resources in cloud environments. Additionally, hardware research platforms have begun the move toward greater integration, providing cache-coherent heterogeneous-ISA processor designs [55, 120] or environments where such processors could be prototyped [37], but rather than offering rapid prototyping, their FPGA/ASIC orientation require long development cycles. As such, these solutions have left open the question of what their accompanying software systems should look like. While we have a number of research operating systems to adopt strategies from [58, 59, 62, 63, 122, 129], none of these had the availability of cache-coherent shared memory, which we argue will make possible a large suite of performance improvements. Indeed, there exist commercial SoCs with some levels of heterogeneity, such as Intel Alder Lake or Arm big.LITTLE. Still, their heterogeneity is more focused on power efficiency than instruction semantics. So, such SoCs simply run traditional OSes for homogeneous-ISA multicores, e.g., vanilla Linux.

A New OS Design. With the emergence of platforms with cache coherent shared memory amongst heterogeneous-ISA processors, which sound like shared memory multiprocessor systems but with heterogeneous cores, a naive question is if also those can run classic SMP OSes? Classic SMP OSes are compiled to run on CPUs of the same ISA, so cannot run amongst heterogeneous-ISA CPUs. Multiple-kernel OSes address ISA-heterogeneity, but existing designs are shared-nothing – i.e., designed to avoid using shared memory between kernel instances, for scalability [62], for heterogeneity, or because cache coherency simply did not exist before [58]. With the introduction of cache coherent shared memory, a new OS design that exploits it together with heterogeneous-ISA is sought: the fused-kernel OS. We implemented such a design atop Linux, in Stramash-Linux, by using components from an academic multiple-kernel OS, Popcorn-Linux [58], to demonstrate the feasibility of our design and its applicability to traditional OSes. While a multiple-kernel OS enables applications to share state, a fused-kernel OS enables applications and kernel code to share (some or all) state among different kernel instances.

Stramash-Linux Stramash-Linux is designed to fully exploit cache-coherent shared memory with the fused-kernel OS design. First and foremost, we nearly eliminate inter-kernel messaging, preferring higher-performance communication via cache-coherent shared memory, including sharing OS' data structures among kernel instances. Based on this, we introduce approaches to make OS services on different kernel instances share data effectively, including locking, because shared data cannot always be shared as-is but may need to be in architecture dependent formats, e.g. page table. While shared data access is the core of the fused-kernel OS design, prototyping Stramash-Linux within the Linux kernel required the introduction of new mechanisms to reduce OS overhead.

Hardware Simulator While we await the envisioned emerging hardware and academic prototypes remain ill-suited to OS development, we implemented our Stramash-QEMU hardware simulator by extending the industry-hardened QEMU with a cache

simulator. By staying in the software realm, Stramash provides for high-speed prototyping exploiting widely-adopted ISAs like x86 and Arm, in contrast to prior hardware-oriented prototyping platforms like BYOC [55] which rely on outdated or less-adopted ISAs.

Stramash-QEMU fuses together multiple QEMU instances with a memory system simulator as the first in a new class of coherent, heterogeneous-ISA simulators. The software flexibility provided by Stramash-QEMU enables the investigation of architecture- and platform-level changes of particular interest to architects as well as research into operating systems. In this chapter, we introduce our fused approach to shared memory and interrupt delivery among others.

Key Results. With our fused-kernel OS, Stramash-Linux, we demonstrate up to $2.6\times$ speedup over the state-of-the-art multiple-kernel OS Popcorn-Linux. And on network serving workloads, up to $12\times$ speedup over Redis. We also validate that Stramash-QEMU can achieve performance measurements within 4% on average of bare metal. Finally, we show that our simulator can be extended for architecture exploration with low effort. Stramash-Linux and Stramash-QEMU will be open sourced upon publish in the academic conference.

Contributions. Briefly, our contributions include:

- The fused-kernel OS design, a shared-mostly multiple-kernel OS that minimises communication overhead among kernel instances, and a prototype of it based on Linux, Stramash-Linux;
- The fused-simulator design, combining single-ISA simulators into a single multi-ISA simulation platform providing cache-coherent shared memory, and its implementation based on QEMU, Stramash-QEMU;
- The validation of Stramash-QEMU using two physical AArch64+x86-64 systems.
- A quantitative comparison of Stramash-Linux versus Popcorn-Linux on Stramash-QEMU.

Limitations and future work. This chapter focuses on a new OS design, its implementation into a state-of-the-practice OS (Linux), and the supporting simulation environment. We deliberately do not study the scalability of this approach – which is not possible to do efficiently today with a software simulator alone. In addition, security and fault-tolerance, while part of our design (see below), are future work. Specifically, we believe we need real hardware to integrate our design with e.g. CXL 2.0 Integrity and Data Encryption functionality. At the time of writing, CXL RAS functionality has not been implemented by the QEMU community.

4.1.1 Design Principles

With the goal of enabling *existing applications* to exploit cache coherent heterogeneous-ISA hardware with maximum performance (with minimal OS/runtime overheads) the fused-OS is based on the following design principles:

- Run applications as-is, or with minimal modifications, to support legacy;

- Target traditional widely-used OS design – monolithic OSes, to attract a large user community, while remaining applicable to other OS designs;
- Generality, to support a wide variety of ISAs, and diverse heterogeneous-memory configurations;
- Minimize data movement; hence, avoid copies across ISA boundaries as much as possible;
- Security.

For our architecture simulator – developed to validate our prototype fused-kernel OS – similar principles apply:

- Exploit, and eventually extend, existing projects, reusing designs and APIs;
- Based on traditional widely-used simulators/emulators to attract a larger user community;
- Fast prototyping, easy configuration – while supporting different memory hierarchies;
- Fast execution speed, with ability to enable a cycle approximate memory model.
- Memory consistency, allowing atomic instruction between different ISAs.

4.1.2 Fused-kernel Operating Systems Design

We introduce a "fused approach" to multiple-kernel operating system design, pulling together – fusing – OS services running on different kernel instances. This means that (some of) the data structures of one kernel instance can be accessed by the other kernel instance(s) directly via shared memory. Hence, enabling multiple kernel instances to work as one, reducing OS service latency and improving execution speed.

Design and Methodology We introduce a new OS design, the *fused-kernel OS*, cf. right of Fig 4.2. Differently from previous multiple-kernel OSes, which are based on the shared nothing principle, fused-kernel OSes are based on the **principle of shared-mostly** – i.e., different kernel instances do communicate using shared memory or share a single state in shared memory, thus avoiding prior work's communication latencies such as serialization and deserialization of data structures, coordination protocols, message copying, etc. This enables tight coordination between kernels to share hardware and software resources.

OS services in a fused-kernel OS are either: **local** or **global**. Local OS services do not require any communication between kernel instances, while global do. Global OS services are mainly built upon shared memory, but when that is too complicated, inter-kernel message-passing is used, like in a multiple-kernel. As shown in previous literature, message-passing requires an OS service to be rewritten as a distributed service, i.e., using coordination protocols. When using shared memory, we introduce two OS service architectures:

- one in which an OS service is rewritten to adopt a **common data format** in shared memory amongst kernel instances;
- one in which each kernel instance keeps **its own data format**, but the others use *accessor functions* to read/write the original data, including locks.

The latter is necessary when handling architecture-dependent data, like page tables, while the first fits services that are architecture independent. A collection of accessor

functions targeting a specific ISA makes up a *remote CPU driver*.

Based on our design principles and the way OS services communicate, our fused-kernel design further introduces the following architectural choices:

Minimal Resource Provisioning To quickly provide global resources, each kernel instance fully utilizes its own private hardware resources (e.g., memory) when available, and acquires any other shared resources only when needed, returning resources to global allocators when are not needed anymore.

This is in stark contrast to traditional OSes: which discover and initialize all resources available on a machine at boot, ready for later allocation. In our design, while all resources are discovered and initialized, shared resources are maintained in a global pool before being assigned to a kernel instance, for instance, a CXL-based shared memory pool for memory resources. Thus, at boot time a kernel instance is given a minimal amount of resources.

Interrupts and Inter-kernel Notification While (most) hardware resources are globally accessible, each kernel knows of but does not use the entire memory, or all devices, but it needs to always map all interrupts, especially IPIs, for communication.

Single virtual address space To simplify the development of *accessor functions*, and reduce their pointer arithmetic overheads, we introduce a single kernel-level virtual address space among kernel instances – which still allows for part of the kernel-level address space to be private.

Security As heterogeneous SoCs are gaining traction, the number of security issues have been rising steadily [60, 158, 165]. For example, a simple defect in a wifi chip could enable remote code execution on an Android phone [89]. Thus, although not the primary focus of this work, a fused-kernel OS needs to consider the security aspects when running on shared memory on different heterogeneous-ISA cores.

Specifically, we postulate that kernel instances should share only required data structures. Everything else should be in private memory or protected by hardware enforcement, e.g. MPU, MMU, IOMMU, and hardware capabilities. To make hardware protection effective we also propose to pack data structures' data in contiguous physical memory – so, it is simple to categorize and share between kernels.

Applications' Compiler and Linker Applications must be compiled in a way that makes them amenable to migration, such that they can continue executing on another ISA-CPU carrying over the existing application state minus the CPU-state that is converted. Inter-kernel thread migration is offered as an OS service and includes functionalities to show the same application state on different kernel instances. Inter-kernel process migration is simpler because there is no kernel state to be kept consistent after migration. In this work we reuse the open-source Popcorn-Linux Compiler Toolchain [26] to compile applications to run on Stramash OS. We direct interested readers to such project literature [11, 27].

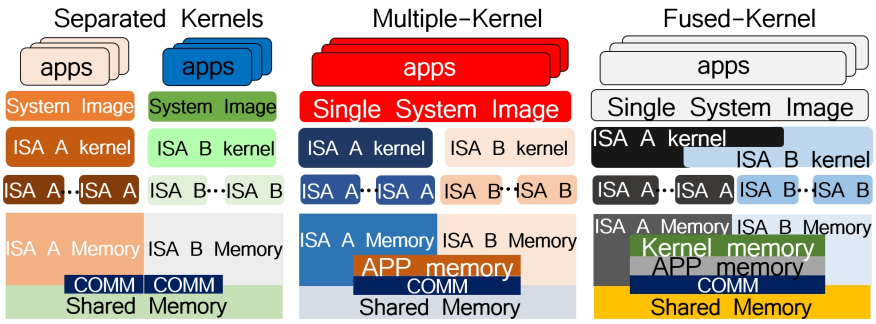


Figure 4.2: Comparison of OS designs for heterogeneous-ISA cache-coherent shared memory platforms.

4.2 Background & Related Work

There have been a variety of architectural studies which have shown performance, efficiency, and security benefits to adopting heterogeneous-ISA system designs [60, 158, 165, 167]. In the systems realm, a number of new operating system [62, 63, 72, 122, 138] designs have been proposed which, with existing or lightly-modified hardware, can exploit the architectural benefits to realise an improved system.

Systems Software. Heterogeneous-ISA platforms with general purpose-CPUs, with or without shared memory, are not new [55, 163]. Hence, scientists and practitioners have come up with different approaches to run software on them. The most common approach is to run a separate software stack – including at least the OS/runtime and applications, per island of homogeneous-ISA CPUs, "separated kernel" in Fig 4.2. Applications must be rewritten to run in a distributed (or offloaded) system where communication happens explicitly via message passing. *This approach cannot exploit cache-coherent shared memory even when available.*

To more effectively support application execution among heterogeneous-ISA cores, new systems software is needed. Applications must be compiled to support live execution migration, and a runtime or OS is needed to provide the source and destination environments. Runtimes and operating systems have both been proposed to support application migration in heterogeneous-ISA platforms. We focus on OS solutions and not on runtimes like H-Container [57] (based on C), and PadMig [91] (based on Java).

In the last decade, several works proposed new OS architectures to provide the abstraction of a single system among different single-ISA or multiple-ISA heterogeneous CPU platforms. Applications running atop these enjoy the same OS interface and services among diverse CPUs, and applications may either spawn threads or migrate threads to a CPU of a different ISA. All works we are aware of achieve this by using multiple, communicating kernel instances, where CPUs of different ISAs run different OS kernel instances, see Fig 4.2. Notable examples include, Helios [138], Barrelfish [62], K2 [122], Popcorn [58], and Flick [72].

Different from SMP OSes, where a single kernel instance runs among several cores, and thus CPU cores share all kernel data structures (shared everything [61]), multiple kernel OSes tend to either not share anything (shared nothing) [58, 62, 72, 138], or share a few kernel data structures (shared something) [121]. Note that shared nothing

multiple kernels do share memory as an optimisation [62]. However, the level of data sharing in kernel space is unrelated to the level of sharing in user space. In fact, shared-nothing multiple kernels do provide applications with consistent shared memory when they run amongst or between kernels, either by distributed shared memory (DSM) [58] or using hardware remapping register (over high-latency PCIe bus) [72].

Two main factors lead the evolution of multiple kernel OSes to shared nothing/something. Firstly, the potentially poor scalability of prior cache-coherent shared memory interconnects [62], or their high power draw [122]. Secondly, the lack of hardware to prototype heterogeneous-ISA CPU cores on cache-coherent shared memory, which eliminates the possibility for such OS development, making our work the first of its kind. Previous projects leverage either non-coherent domains [58, 122], or domains connected via high-latency peripheral buses exploiting snooping [72].

Prototyping Environments Earlier heterogeneous-ISA system prototypes have largely exploited existing hardware. A number of systems used existing x86 server hardware with PCIe non-transparent bridges to connect processors of other ISAs [63]. Others exploited existing systems-on-chip which provided limited heterogeneous-ISA capability (e.g. Arm+Thumb) [122]. While these platforms enable full-speed execution of proposed OS designs, they do not feature cache-coherent memory, necessitating software coherence layers. These degrade performance and increase OS complexity, particularly for research systems trying to build on top of Linux.

Recently, BYOC introduced a cache coherent heterogeneous-ISA prototyping environment [55]. BYOC uses FPGA emulation to provide high-speed prototyping which could eventually be realised in silicon. The system supports several ISAs (SPARC v9, RISC-V, i486), but does not feature 64 bit x86 and Arm ISAs, which would be of most interest to OS developers and users. Further, prototyping an architectural change requires a full, FPGA-ready implementation, which is a high bar to entry. Lately, AMD announced an Embedded series SOC that has Ryzen + Xilinx FPGA [3], SOPHON's SG200X series are new SoCs featuring RISC-V and Arm cores [13], and architecting such chips for HPC is inevitable in the cloud [4].

We take a different approach to prototyping by fusing multiple instances of QEMU with an extension of the Cache plugin memory system simulator. This enables OS, architecture, and platform prototyping in software at a relatively high speed without the need for full hardware implementation of proposed modifications. With QEMU, we can also exploit proprietary ISAs like 64-bit x86 and Arm. Our work is the first we are aware of to fuse multiple QEMU instances of different ISAs together with shared memory backed by an architectural cache simulator.

4.3 Hardware Model

Stramash targets emerging platforms with cache-coherent heterogeneous-ISA CPUs integrated into an SoC (like BYOC, or Milk-V Duo), or interconnected via emerging cache-coherent buses (like CXL). Hence, we considered at least 3 memory hardware configurations as depicted in Fig 4.3: *Separated*, *Shared*, and *Fully Shared*. In the *Separated* configuration each CPU group has its own memory, with coherence managed by a Last Level Cache (LLC), like in NUMA. In the *Shared* configuration, each

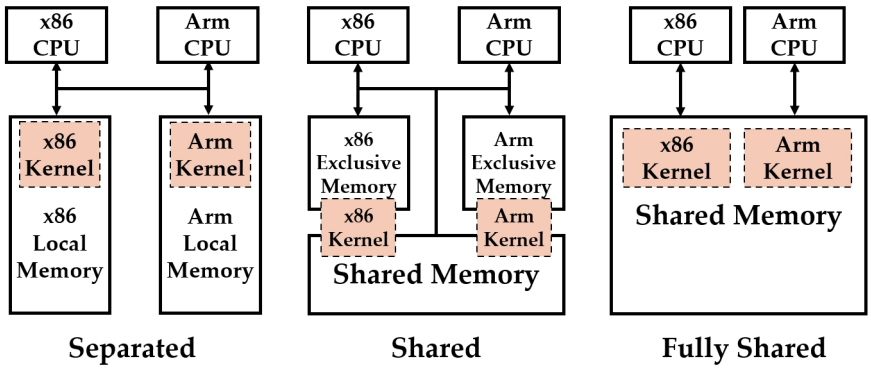


Figure 4.3: Different memory configurations and OS kernel placement.

CPU group has access to a private memory, and all CPU can access a shared memory, like CXL 3.0 [75]. In the *Fully Shared* configuration there is a single, shared memory for all processors, although it may be mapped to different addresses, like academic research project: Openpiton [56]. Each processor is capable of sending Inter Processor Interrupts (IPI) to any other processor in the system. All MMIO devices are accessible by all processors. Regarding memory consistency, we assume all processors abide by the strongest memory consistency model of all ISAs (Arm already supports running in TSO mode). This assumption is justified by standards like CXL 3.0 that adds a MESI cache coherency protocol for inter-host shared memory – where hosts can be of any ISA. Systems supporting heterogeneous consistency models could take advantage of tools like ArMOR [127] to defend against consistency model mismatches.

Beyond existing heterogeneous-ISA systems' capability to communicate via interrupts, I/O devices, and non-coherent shared memory, Stramash leverages cache coherent shared memory for communication. For inter-CPU-group communication, traditional systems relied on mechanisms like custom communication devices (e.g. hardware FIFOs or spinlocks [163]) in embedded SoCs, or in servers, the use of built-in switching functionality in an Ethernet NIC [66, 131], i.e., sending network packets between ISAs. We use the latter, common mechanism as a baseline.

4.4 Design Principles

With the goal of enabling *existing applications* to exploit cache coherent heterogeneous-ISA hardware with maximum performance (with minimal OS/runtime overheads) the fused-OS is based on the following design principles:

- Run applications as-is, or with minimal modifications, to support legacy;
- Target traditional widely-used OS design – monolithic OSes, to attract a large user community, while remaining applicable to other OS designs;
- Generality, to support a wide variety of ISAs, and diverse heterogeneous-memory configurations;
- Minimize data movement; hence, avoid copies across ISA boundaries as much as possible;
- Security.

For our architecture simulator – developed to validate our prototype fused-kernel

OS – similar principles apply:

- Exploit, and eventually extend, existing projects, reusing designs and APIs;
- Based on traditional widely-used simulators/emulators to attract a larger user community;
- Fast prototyping, easy configuration – while supporting different memory hierarchies;
- Fast execution speed, with ability to enable a cycle approximate memory model.
- Memory consistency, allowing atomic instruction between different ISAs.

4.5 Fused-kernel Operating Systems Design

We introduce a "fused approach" to multiple-kernel operating system design, pulling together – fusing – OS services running on different kernel instances. This means that (some of) the data structures of one kernel instance can be accessed by the other kernel instance(s) directly via shared memory. Hence, enabling multiple kernel instances to work as one, reducing OS service latency and improving execution speed.

Design and Methodology We introduce a new OS design, the *fused-kernel OS*, cf. right of Fig 4.2. Differently from previous multiple-kernel OSes, which are based on the shared nothing principle, fused-kernel OSes are based on the **principle of shared-mostly** – i.e., different kernel instances do communicate using shared memory or share a single state in shared memory, thus avoiding prior work's communication latencies such as serialization and deserialization of data structures, coordination protocols, message copying, etc. This enables tight coordination between kernels to share hardware and software resources.

OS services in a fused-kernel OS are either: **local** or **global**. Local OS services do not require any communication between kernel instances, while global do. Global OS services are mainly built upon shared memory, but when that is too complicated, inter-kernel message-passing is used, like in a multiple-kernel. As shown in previous literature, message-passing requires an OS service to be rewritten as a distributed service, i.e., using coordination protocols. When using shared memory, we introduce two OS service architectures:

- one in which an OS service is rewritten to adopt a **common data format** in shared memory amongst kernel instances;
- one in which each kernel instance keeps **its own data format**, but the others use *accessor functions* to read/write the original data, including locks.

The latter is necessary when handling architecture-dependent data, like page tables, while the first fits services that are architecture independent. A collection of accessor functions targeting a specific ISA makes up a *remote CPU driver*.

Based on our design principles and the way OS services communicate, our fused-kernel design further introduces the following architectural choices:

Minimal Resource Provisioning To quickly provide global resources, each kernel instance fully utilizes its own private hardware resources (e.g., memory) when available, and acquires any other shared resources only when needed, returning resources to global allocators when are not needed anymore.

This is in stark contrast to traditional OSes: which discover and initialize all resources available on a machine at boot, ready for later allocation. In our design, while all resources are discovered and initialized, shared resources are maintained in a global pool before being assigned to a kernel instance, for instance, a CXL-based shared memory pool for memory resources. Thus, at boot time a kernel instance is given a minimal amount of resources.

Interrupts and Inter-kernel Notification While (most) hardware resources are globally accessible, each kernel knows of but does not use the entire memory, or all devices, but it needs to always map all interrupts, especially IPIs, for communication.

Single virtual address space To simplify the development of *accessor functions*, and reduce their pointer arithmetic overheads, we introduce a single kernel-level virtual address space among kernel instances – which still allows for part of the kernel-level address space to be private.

Security As heterogeneous SoCs are gaining traction, the number of security issues have been rising steadily [60, 158, 165]. For example, a simple defect in a wifi chip could enable remote code execution on an Android phone [89]. Thus, although not the primary focus of this work, a fused-kernel OS needs to consider the security aspects when running on shared memory on different heterogeneous-ISA cores.

Specifically, we postulate that kernel instances should share only required data structures. Everything else should be in private memory or protected by hardware enforcement, e.g. MPU, MMU, IOMMU, and hardware capabilities. To make hardware protection effective we also propose to pack data structures' data in contiguous physical memory – so, it is simple to categorize and share between kernels.

Applications' Compiler and Linker Applications must be compiled in a way that makes them amenable to migration, such that they can continue executing on another ISA-CPU carrying over the existing application state minus the CPU-state that is converted. Inter-kernel thread migration is offered as an OS service and includes functionalities to show the same application state on different kernel instances. Inter-kernel process migration is simpler because there is no kernel state to be kept consistent after migration. In this work we reuse the open-source Popcorn-Linux Compiler Toolchain [26] to compile applications to run on Stramash OS. We direct interested readers to such project literature [11, 27].

4.6 Stramash-Linux Implementation

We implemented a prototype of our fused-kernel OS, called Stramash-Linux, based on Linux kernel 5.2.12. To avoid reinventing the wheel, we adopted OS components from the open-source Popcorn-Linux project including process/thread migration and the messaging layer. To build Stramash-Linux we contributed around 9200 LoC atop

Popcorn-Linux. We chose the Popcorn-Linux project because it is the only one providing an open-source, fully-functioning OS and compiler toolchain to produce executable binaries that can migrate across heterogeneous-ISA CPUs.

Prototype Limitations Because we based our work on the Popcorn project, which fully supports only the x86 and Arm ISAs, our Stramash prototype inherits the same limitation. However, we believe that our design applies to other ISA mixtures as well, including different bit-width and potentially endianesses. The primary issues in supporting those mixtures lays is the engineering effort. x86-64 and AArch64 are widely adopted, and specifically on emerging platforms targeted by this work. Hence, other ISAs mixtures are out of the scope of this work. While we did implement support for data packing in contiguous physical memory – including moving pages to reorganize data, we did not find an efficient method to limit the remotely accessed memory in kernel space on different architectures. Capabilities could be a potential candidate, but this is left as future work.

4.6.1 Kernels Booting

Within the hardware model proposed, heterogeneous ISA cores would have access to the entire shared memory, devices, etc. Therefore, like a traditional OS, Stramah-Linux will discover all memory and devices, but initialize only a minimal set of those to enable a working system, the rest of the resources are managed by global pools. At the time of writing, we limit the area usable by each kernel instance using BIOS tables/device trees. The OS reads the memory map tables provided by the firmware and adjusts its boundaries based on that. Thus, kernel instances' memory areas do not overlap (see Fig 4.4. Once the boot is complete, kernel instances establish a communication channel to coordinate and bring up all OS services to share resources in a fused manner.

4.6.2 Message-passing Communication

Stramash-Linux uses a messaging layer to communicate between kernel instances. This is based on one or more pairs of shared memory ring buffers per kernel pair, to exploit shared memory to minimise latency and payload cost. After a message has been enqueued on a ring buffer, the messaging layer sends a cross-ISA IPI to the receiver core group. We also support polling in place of interrupt dispatching.

4.6.3 Global Memory Allocator

Stramash-Linux implements a global memory allocator that manages physical memory amongst kernel instances.

The current version memory allocator exploits and extends the memory Hot-plug subsystem, with certain modifications. Different from Hot-plug, Stramash-Linux does not require the memory block to be unplugged. Instead, for hot removal it first evacuates the memory block and then isolates the pages. Although we experimented with different options, including the continuous memory allocator (CMA) as in K2 [122], all require major modifications to the Linux kernel source code.

The Memory Allocator. Fig 4.4. shows an example of memory layout in Stramash on an Arm+x86 configuration with 8GB of RAM. We have implemented a fixed-size global memory allocator with a configurable block size (from 32MB to 4GB). we opted for a minimum size of 32MB to reduce the overhead associated with frequent memory assignments. Each kernel starts with a fixed amount of blocks. When the total memory pressure on a certain kernel instance passes 70%, that kernel requests an additional block from the global memory allocator. If a block is free, it is directly assigned. If there are no free blocks, the allocator will try to evict a block from the other kernel until it also reaches the same memory pressure.

4.6.4 Fused Virtual Address Space

Stramash-Linux supports user-space process/thread migration across ISAs. Unlike Popcorn-Linux, which uses software DSM to provide a single virtual address space among kernels (which passes entire memory pages as messages), Stramash-Linux leverages cache coherent shared memory to maintain a page table per kernel instance due to the differing page table format in Arm and x86. Both page tables refer to the same physical memory pages for the same application.

Fused virtual address space: Stramash-Linux aligns kernel virtual addresses across different kernel instances, enabling full addressability of another kernel's memory. By adjusting the vmalloc ranges of x86 to align with the direct map range of the Arm instance, the Arm's virtual address space becomes fully addressable to the x86 kernel instance, and vice versa. We refer to this configuration as the Fused Virtual Address Space. This distinctive capability distinguishes Stramash from other multi-kernel operating systems by allowing it to share kernel virtual addresses and data structures seamlessly, enhancing interactions without redundancy.

It is worth noting that for some data structures, today we need to disable the randomized layout to enable direct remote access. If the layout of shared data structures varies, some (simple) handling is needed. However, this is typically not a concern, as few data structures vary across ISAs.

Software Remote Page Table Walker. Stramash-Linux allows one kernel to access the page table of the other kernel through a cross-ISA software page walk to reduce long-latency round-trips message passing overhead. Currently, both x86 and Arm in Stramash-Linux are using 5 level page tables. To acquire the page table entry (PTE) in the origin kernel, the remote kernel has to walk through those tables with proper page masks. Each level page mask is re-defined if it is different between x86 and Arm.

Software Remote VMA Walker. Unlike Popcorn-Linux, where a VMA fault triggers a message exchange to the original kernel, in Stramash-Linux, each kernel can access the other kernel's VMA lists, with appropriate VMA locks acquired. Note that in our current implementation, the VMA lists are still maintained using the RB-tree structure.

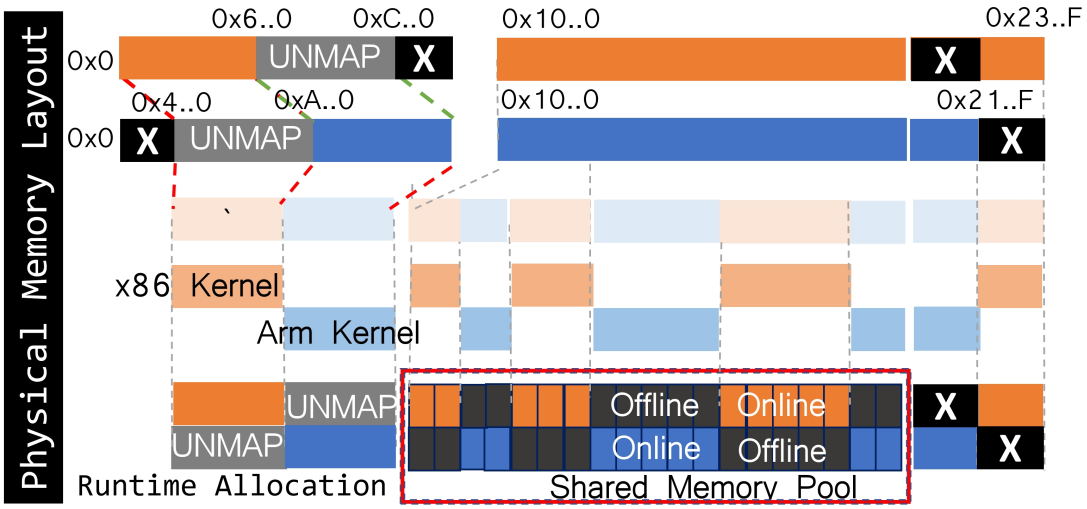


Figure 4.4: Physical memory layout example. x86 instance starts at `0x0`; Arm instance starts at `0xA00000000`; Shared Memory Pool from `0x100000000` to `0x200000000`.

Stramash(fused-kernel) Page Fault Handler. In the current version of Popcorn-Linux, anonymous pages are allocated in the origin kernel – where the application starts, which introduces at least 2 rounds of message passing, i.e., the request/response of the page allocation and replication.

In Stramash-Linux, the remote kernel allocates anonymous pages without needing to notify the origin kernel. The remote kernel first allocates a page, inserts it into its own page table, and adds it to the origin kernel's page table with the remote node ISA format. Once the process migrates back to the origin kernel, the origin kernel can simply reconfigure the PTE to its own format and access the page via cache-coherent shared memory. When a process is terminated, the origin kernel only invalidates the PTE and does not attempt to release the page, as it was allocated by the remote kernel. The remote kernel, on the other hand, takes responsibility for invalidating its own PTE and releasing the page, thus finalizing the memory recycling. To manage simultaneous access, we have implemented a cross-ISA page table lock `(Stramash-PTL)`, ensuring that only one instance can modify the page table at a time. This method effectively bypasses the reliance on the Copy-On-Write (COW) policy heavily utilized in Popcorn-Linux. In contrast to Popcorn, where the COW policy minimizes messaging during frequent page updates, Stramash-Linux allows direct page access without such overhead.

4.6.5 Cross-ISA locking

Atomicity Stramash-Linux's AArch64 kernel includes support for Large System Extensions (LSE) [16], which provides a non-interruptible read-modify-write sequence in a single instruction, Compare-and-Swap (CAS). These atomic instructions can replace Load-Link/Store-Conditional (LL/SC) operations. Stramash ensures that all kernel spinlock-related instructions use the CAS instruction, providing a more efficient and robust mechanism for handling cross-ISA locks. Additionally, with the integration

of Stramash-QEMU, detailed in Sec 4.7.1, operations involving atomic instructions maintain their integrity cross-ISAs.

Futex Popcorn-Linux manages Futex (fast userspace mutex) operations by relying on the origin kernel to create and control all Futex instances. When a lock is requested, the remote kernel must message the origin kernel to engage the lock, and all subsequent locking actions are maintained by the origin kernel. In contrast, Stramash-Linux allows the remote kernel direct access to the Futex locking list. This direct access streamlines the process, reducing dependency on the origin kernel for locking operations. Upon unlocking, if the thread is currently waiting in the origin kernel, the remote kernel sends a cross-ISA IPI to the origin kernel to wake up the thread.

4.6.6 Fused Namespace

For applications that migrates inter-ISA Stramash-Linux enables the same mnt, PID, net, UTS, user, and cgroup namespaces. These provide the same environment when an application migrates. Moreover the same list of CPU including topological information is available on every kernel instance.

4.7 Stramash Hardware Simulator

A hardware simulator is necessary to thoroughly evaluate our Fused-kernel OS against state-of-the-art OSes for emerging platforms. Inspired by the Fused-kernel OS, our *fused-simulator* connects multiple traditional single-ISA simulators as one and presents cache coherent shared memory to every simulated core. Stramash-QEMU also supports mechanisms for communication across ISAs other than shared memory, including IPIs, memory remapping, private memory, device sharing, parallel bootup, and Cache simulator. Stramash-QEMU targets emerging platforms with cache-coherent shared memory, focusing primarily on memory system modeling. It is worth noting that Stramash-QEMU is generic enough to be reconfigured for different hardware models. We based our simulator on QEMU 8.0.0 to execute software on heterogeneous CPU cores, specifically AArch64 and x86-64. We have extended the current QEMU Cache plugin [8] to support a 3-level cache plus CXL. Our contributions amount to approximately 7100 LoC to QEMU.

We employ system-level simulation to model all OS and application details. As the application runs, it accesses a main memory that is coherent across all simulator instances. With Cache simulation enabled, all memory accesses are forwarded to our Cache plugin, which provides detailed memory access overhead and feedback on latency to our timing model.

4.7.1 Pervasive Cache-coherent Shared Memory

In Stramash-QEMU, guest memory is allocated on a per-host basis. Any memory operation from a single guest will be reflected in others, respecting the rules of cache coherency. By running both Stramash-QEMU instances on the x86 host, the actual memory operations will follow the host's x86 TSO memory consistency model. Given

that this is a stricter model, this setup ensures that both instances are protected from memory consistency issues. However, there is a potential complication where the x86 host translates the Arm guest's LL/SC instructions into the host's CAS instructions [77, 127]. Stramash-QEMU uses the Cortex-A76 as the Arm CPU core, which supports LSE and thus CAS instructions. We have carefully configured the QEMU code generator (TCG) to ensure relevant Arm instructions are correctly translated.

4.7.2 Inter-ISA Interrupts

Interrupts are crucial for interactions between CPU cores and between cores and external devices. In a heterogeneous-ISA setting, different ISAs have their approaches to interrupt management. To take a fused approach to interrupt management and thus facilitate interrupt sharing across architectures, we have prototyped cross-ISA IPIs that enable native IPI communication between CPUs of different ISAs. We extended the AArch64 SGI and x86 APIC by adding extra logic to route the native IPI to the peripheral device, and then notify the other ISA to generate a native IPI. We assign an unused IRQ number in Linux and set up respective handlers for each ISA's kernel to handle the IPI.

4.7.3 Stramash Timebase

Stramash-QEMU introduces a cross-ISA timing model to coordinate simulated time. Our design resembles manycore simulators such as PriME [86], where the memory system becomes the primary factor, and core performance is, by default, modeled with a fixed non-memory IPC [87]. This allows us to implement our custom Stramash time base, which can quickly simulate cycle-accurate timing information.

4.7.3.1 Time in QEMU

QEMU is an instruction simulator rather than a cycle-accurate simulator. While QEMU can provide the guest with emulated time, this measurement is rather basic and does not offer much insight into the performance of the emulated hardware. Additionally, QEMU's TCG backend supports instruction counting (icount), enabling the counting of executed instructions. Icount is a simple yet effective way to profile software. It is widely supported in tools such as perf, Valgrind, and Intel Pin [126]. We have configured QEMU to utilize an instruction count-based timing model, with time progressing according to instruction counting. This ensures alignment in speed between I/O and instruction emulation, preventing an emulated hardware device from running unrealistically fast. We have disabled the warp time feature of QEMU to eliminate any influence from the host's real-time adjustments.

4.7.3.2 Instruction Counting

Measuring programs' execution time in a heterogeneous-ISA platform is not as straightforward as in traditional homogeneous-ISA platforms because the application can migrate between ISA CPUs at runtime. We have integrated our icount approach with

Linux Perf to get an accurate measurement of the time that the application has actually executed. We use this approach in Stramash-QEMU validation, comparing our instruction count to the native instruction count on the physical machine, described in Section 4.9.1.2.

4.7.3.3 Timing Modelling

Although the alignment of the instruction count-based timing model ensures a minimum latency for each emulated instruction, it does not offer any performance modeling metrics. To address this limitation and enable system-level profiling, we have integrated a modified version of QEMU's Cache plugin. We have added a 3-level cache feature to model the performance of memory systems, which has been validated in comparison with the GEM5 MESI three-level cache memory model [28] in Sec 4.9.1.3. Each memory instruction executed by QEMU passes to our Cache plugin which analyzes the cache behavior, whether it is a hit or miss, and accordingly adds memory access overhead to the icount counter. This information is then sent back to QEMU as feedback, enhancing the accuracy of the performance modeling metrics.

Additionally, we have defined different memory models shown in Fig 4.3. Based on the fault address, when an L3 cache miss occurs, different overheads are generated and fed back to Stramash-QEMU. In the `Fully Shared`, all memory accesses are considered local. In contrast, in the `Shared` and `Separated`, remote memory access overhead is added if the fault address falls within the range defined as remote memory. The timing information within the guest mirrors the changes modeled by our QEMU Cache plugin. Our Cache plugin access overhead can be found in Table 4.2.

4.7.3.4 CXL access overhead feedback

Stramash-QEMU Cache plugin simulates the latency of data transmission on the CXL bus. We model the overhead introduced by CXL to maintain coherence among replicas in the caches of various processors in heterogeneous systems. We consider the additional delays brought on by SNOOP messages and Responses, which play a crucial role in the invalidation and update processes, including Back-Invalidate Snoop, Snoop Invalidate, and Snoop Data [75]. When one processor attempts to write to the same memory location accessed by another, it issues a "Snoop Invalidate" request. This operation mandates that all other processors invalidate the corresponding cache lines they hold, ensuring coherence and preventing outdated data access. Conversely, if a processor intends to read from the same memory location already accessed by another, it triggers a "Snoop Data" request. This command converts the cache line's state from "Exclusive" to "Shared" in other processors.

4.7.4 IO Devices

We have enhanced QEMU such that when an instance lacks a particular device, it creates a memory mapping for that device. Consequently, all memory accesses are redirected to the QEMU instance containing the respective device.

Table 4.1: Machines for Popcorn-Linux baseline data collection

Name	Core	Hz	RAM
Small_Arm	Broadcom Armv8 A72 8 cores	3.0GHz	8GB
Big_Arm	Dual Cavium ThunderX2 CN9980 v2.2 (32 cores/128 threads)	2.0GHz	256GB
Small_x86	Xeon E5-2620 v4 (8 cores/16 threads)	2.1GHz	16GB
Big_x86	Dual Xeon Gold 6230R (26 cores/52 threads)	2.1GHz	768GB

Table 4.2: Stramash-QEMU Cache plugin Memory Operation Latency, CXL latency for remote memory[157]

Core	Operation	Latency(cycles)
Cortex-A72[5]	L1/L2/L3/mem/remote-mem	4/9/*/300/780
ThunderX2[6]	L1/L2/L3/mem/remote-mem	4/9/30/300/620
E5-2620[15]	L1/L2/L3/mem/remote-mem	4/12/38/300/640
Xeon Gold[9]	L1/L2/L3/mem/remote-mem	4/14/50/300/640

4.8 Experimental Methodology

We evaluated Stramash on a Dell PowerEdge R440 server with Intel Xeon Silver 4110 (8 cores/16 threads) at 2.1GHz and 64GB of RAM. We setup Stramash-Linux on the different hardware models described in Sec 4.3 and compare it with Popcorn-Linux, which we setup using both network-based and shared-memory-based messaging layers. To the best of our knowledge, there are no other open-source projects that enable applications to run across different ISAs aside from Popcorn-Linux. Note that our focus is not on the performance and power benefits of heterogeneous-ISA versus homogeneous-ISA systems: those have already been extensively discussed in previous works. Instead, we aim at exploring applications' behaviour when running Stramash-Linux or Popcorn-Linux on different CC shared memory hardware models – all simulated using our Stramash-QEMU.

4.8.1 Stramash-QEMU Setup

Our Stramash-QEMU simulates various hardware models, herein we focus on the three models in Fig 4.3, which we configure with different physical memory layouts for each QEMU instance, see Fig 4.4 as a reference. In the **Separated** model, the x86 instance has local memory ranges 0x0 – 0x5FFFFFFF, 0x100000000 – 0x14FFFFFFF, and 0x220000000 – 0x23FFFFFFF, and the Arm instance has local memory ranges 0xA0000000 – 0xFFFFFFFF and 0x15000000 – 0x21FFFFFFF. In the **Shared** model, the memory range 0x100000000 – 0x200000000 is considered as remote memory for both x86 and Arm instances, with all other local memory ranges remaining the same as in the **Separated** model. In the **Fully Shared** model, all memory ranges are considered local memory. In the experiments, when not noted differently, we use the memory latencies from Xeon Gold and ThuderX2 pairs as shown in Table 4.2.

When Stramash-QEMU encounters a memory operation to an address, it first checks if the address is in the cache and then it adds the corresponding cache latency as specified in Table 4.4. If the cache line is invalidated or miss, it reloads it from memory back into the cache. Based on different hardware models, the address could be in local memory or remote memory, and the corresponding latency is added accordingly. Therefore, in the **Shared** model if an address is found in another cache, depending on whether it's a read or write operation, the Cache-Plugin will follow the appropriate MESI transition and adds the simulated CXL snooping overhead. For example, if one node writes to a cache line, the other node will invalidate that cache line if it is in shared state through a simulated invalidation – a snoop invalidation overhead is added. The **Separated** model, could be configured as NUMA or CXL, currently we use the CXL snooping overhead to simulate the cost of cache coherence, but it can be set with the cost of Intel QPI[104] or AMD Infinity Fabric[2], etc. **Fully Shared** includes just one shared cache and memory. After such operations have been simulated, we add the corresponding overheads to the QEMU icount. We also actively maintain the same icount speed on both QEMU instances to ensure that both QEMU icounts increase and proceed at a similar rate.

Setup for Stramash-Linux. It is important to note that local and remote physical memory ranges are fixed for both the x86 and Arm instances in each hardware model simulated by Stramash-QEMU. However, Stramash-Linux's global allocator can adjust the memory capacity of each kernel instance, but without altering the layout of the physical memory. If the x86 instance acquires a physical memory range that belongs to the Arm's local memory, any memory operations to this physical memory from the x86 will incur additional remote memory access overhead, simulated by our Cache-Plugin.

Setup for Popcorn-Linux. Unlike Stramash-Linux, Popcorn-Linux kernel instance *do not* directly access each other memory. We run two versions of Popcorn-Linux.

The first, Popcorn-Linux Messaging over Shared Memory (**SHM**) employs a shared memory–based messaging layer. We provide a shared memory area of 128 MB in the range 0x50000000 – 0x58000000 for x86 and 0x90000000 – 0x98000000 for Arm. Thus, when either side accesses the physical memory within this shared ranges, based on the corresponding hardware model, the Cache Plugin will add corresponding memory access latency. Meanwhile, all other physical memory is exclusive used by one or the other kernel. We setup **Popcorn SHM** on all 3 hardware models, **Separated-SHM**: the messaging layer is mapped at x86's local and Arm's remote memory; **Shared-SHM**: the messaging layer is mapped at both kernel's remote memory; **Fully Shared-SHM**: the messaging layer is mapped at both kernel's local memory. For Stramash-Linux and Popcorn-Linux messaging over shared memory, the IPI overhead is 2 microseconds, as mentioned in Sec 4.9.1.1.

The second, Popcorn Linux Messaging over Network (**TCP**) where each kernel only accesses its local memory and communicates with other kernels via `TCP/IP`. Because this doesn't exploit shared memory, it performs the same independently of the hardware model. We add approximately 75us delay for each message round trip to simulate network latency, which is the average latency of a 64KB (Default Popcorn-Linux

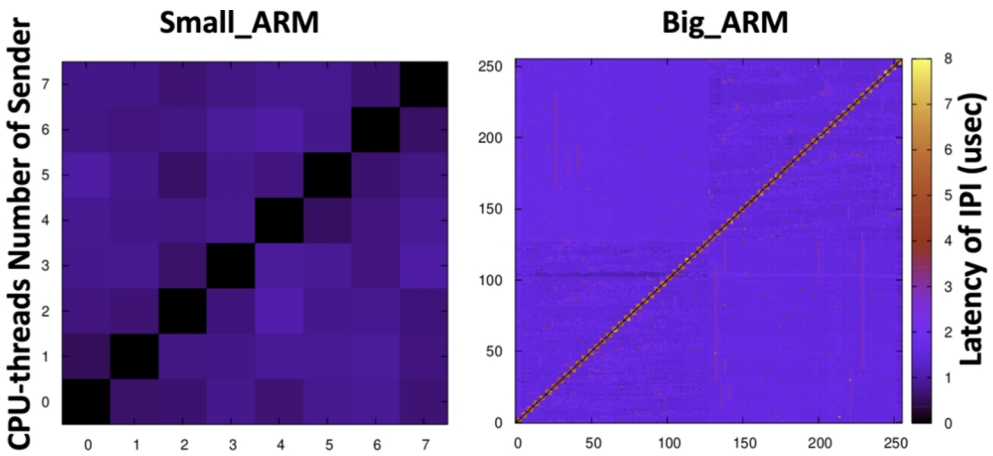


Figure 4.5: IPI latency results Arm

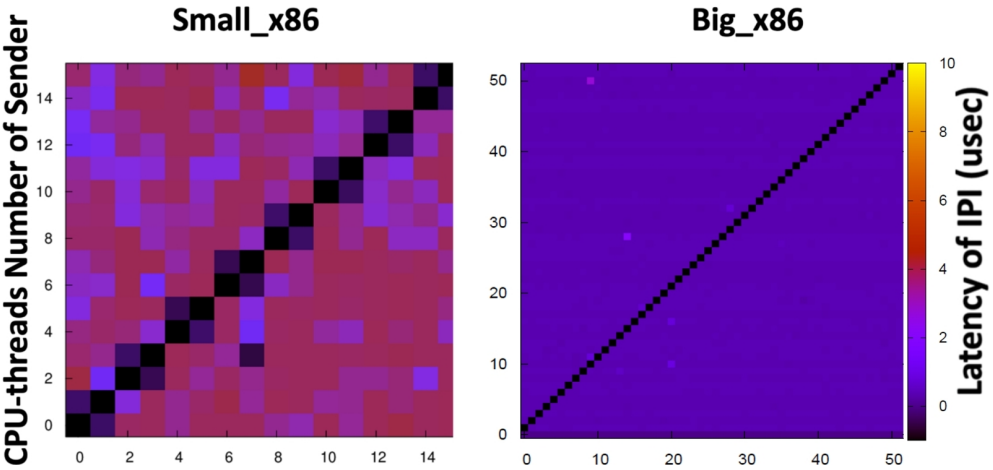


Figure 4.6: IPI latency results x86

message payload size [31]) packet round trip time measured software-to-software on SmartNIC hardware [66].

4.8.2 Benchmarks.

Amongst others, we extensively run workloads from the NAS Parallel Benchmark (NPB) [135] collecting execution time, instructions, and cycles number on bare-metal and on Stramash-QEMU – using our Linux perf. We selected NPB because it has different memory access patterns, including read and write intensive workloads. We used different NPB benchmark classes depending on the experiment.

4.9 Evaluation

4.9.1 Stramash-QEMU Validation

We validated the accuracy of our hardware simulator, Stramash-QEMU, with bare-metal measurements on the two reference platforms. Those are two pairs of Arm

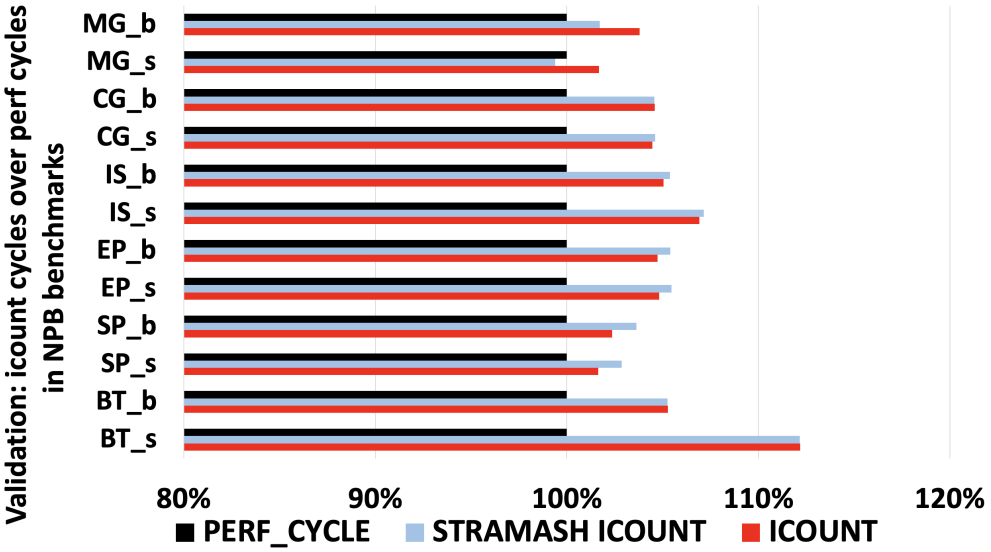


Figure 4.7: icount validation of small_x86 and small_Arm ($*\_s$), and also big_x86 and big_Arm ($*\_b$). Note relative error *x axes*, always $< 13\%$.

and x86 machines, *small_Arm* and *small_x86*, *big_Arm* and *big_x86*. The small _Arm, a smartNIC, and the small_x86, a low-spec server, are interconnected via PCIe bus where the smartNIC is plugged in. To run Popcorn-Linux on this setup we ported it to the Linux kernel 4.14.79 needed by the smartNIC. The big_Arm and big_x86, two high-spec servers, are running Popcorn-Linux kernel version 5.2.12. The servers are connected through 100Gbps Ethernet. In both cases, Popcorn-Linux exploits the `TCP/IP` messaging layer. Their technical details are shown in Table 4.1. We also compare Stramash-QEMU Cache-Plugin with Gem5 to assess the rigorousness of our cache model.

4.9.1.1 IPI Cost Characterisation.

Stramash-QEMU integrates two or more QEMU instances, each emulating a different ISA processor complex. To the best of our knowledge, we are the first to enable QEMUs to exchange cross-ISA IPIs. In the absence of real hardware to measure our proposed cross-ISA IPI, the exact latency remains unknown. Therefore, we used the latency of cross-NUMA IPI as a placeholder for cross-ISA latency — a configurable parameter in Stramash. To obtain a realistic IPI overhead, we measured IPI latency on real machines.

We implemented a kernel module to measure IPI latency between all core pairs on both Arm and x86 machine sets with minimal overhead. The RDTSC instruction was used to capture timestamps between the IPI sending and receiving. We used MONITOR/MWAIT to minimize measurement overhead. Fig 4.5 and Fig 4.6 show the measurement results. The average IPI latency is about 2 microseconds in large machine pairs, and we have used this value as our simulated cross-ISA IPI cost.

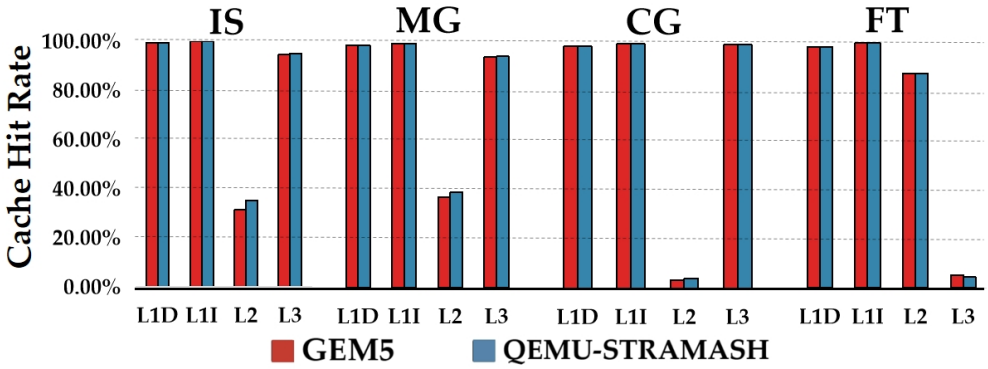


Figure 4.8: Cache Simulation Validation

4.9.1.2 Instruction Count.

We use NPB benchmarks for icount validation. We use the Stramash-QEMU perf+icount tool introduced in Sec 4.7.3.2 to record the icount data from both kernel instances during the migration. By comparing the instruction per cycle (IPC) results from the (native) perf tool on a real machine, we estimate the IPC for Stramash. Since Stramash-Linux target heterogeneous platform is not available, we use Popcorn-Linux on two real machines and native perf to profile the application pre- and post-migration, collecting instruction and cycle counts. We then align these native perf results with the Stramash icount data to approximate the cycles. Then we compare this approximation with the actual cycle counts from native perf on the two machines.

The results are shown in Fig 4.7, the suffix b and s represent the native perf on big machine set and small machine set shown in Table 4.1. The `ICOUNT` bars are the Popcorn messaging with `Shared Memory` approximated cycle result, the `STRAMASH ICOUNT` bars are the result with fused virtual address enabled, while the `PERF_CYCLE` bars are the native perf result.

The overheads of icount are always less than 13%, and about 4% on average – hence, the accuracy of the tool.

4.9.1.3 Cache plugin.

Stramash-QEMU, paired with the Cache plugin, targets precise timing verification as mentioned in Sec 4.7.3.3. We compare our Cache plugin results with the Gem5 MESI Three-Level cache model, a Ruby modular framework for building cache coherence protocols in simulation environments. The same three-level cache structure and size are constructed in our Cache plugin to ensure consistency in our comparisons. We evaluated the NPB benchmarks—CG, IS, MG, and FT—and analyze their cache behaviur. The results are presented in Fig 4.8. It illustrates the cache hit rates for different cache levels, including the L1 instruction/data cache, L2, and L3 caches. Across the four benchmarks, the performance of our cache simulator closely aligns with that of Gem5 Ruby, with discrepancies in L1, L2, and L3 caches being less than 5%. This demonstrates the accuracy of our cache simulator.

4.9.2 Stramash-Linux Evaluation

We conducted several experiments to demonstrate the difference between Stramash-Linux with Popcorn-Linux to answer the question: Does Fuesd-kernel OS design is better than Multi-kernel OS, or is there a tradeoff? In the following experiments, we set up a pair of AArch64 and x86-64 kernel instances in Stramash-QEMU, with 8GB of memory in total. We run NPB application that migrates between CPUs, the migration points choose are same as the Popcorn Linux [58], all processing procedure is offloaded.

4.9.2.1 Benchmarking Cross-ISA Migration.

We executed various NPB benchmarks and measured their execution times within Stramash-QEMU using our Cache plugin's timebase. The results are shown in Fig 4.9, with the y-axis representing execution time (lower is better) normalized to the Vanilla case (the application runs locally without migration involved). Popcorn-Linux Messaging over Network (**TCP**) and Popcorn-Linux Messaging over Shared Memory (**SHM** with 3 different memory models) represent the multi-kernel solution and serve as our baseline. **Fully Shared**, **Shared** and **Separated** shows the results of Stramash-Linux with the specific memory models used in Stramash-QEMU, recall in Fig 4.3. We observed that Stramash Linux results provides up to a $2\times$ speedup compared to Popcorn-Linux **SHM** with same memory model set up, and $2.6\times$ to **TCP** in IS benchmark. However, some benchmarks show less significant speedup.

Performance improvement breakdown. In order to explore the reason why the performance varies to different application, we breakdown the overhead of each NPB benchmark, focusing on three main components: the messaging costs through different message layer (`MSG`), the memory access overhead (`Local` or `Remote`), and the plain instructions (`INST`). From our observation, the performance varies across different NPB benchmarks, indicating that performance improvement is application-specific. In all cases, Stramash Linux with (**Fully Shared**) memory model demonstrates the best performance, closely matching that of the Vanilla case, as it effectively eliminates remote memory access and messaging overheads. The use of IPC 1 in our Stramash-QEMU might obscure the cost of additional plain instructions introduced by Stramash-Linux, but as we can see, this cost is negligible.

Message Passing Overheads. Based on the Popcorn Linux results, the major difference between the two models (**SHM** and **TCP**) arises from message passing overhead, as we added extra network latency to simulate the `TCP/IP` networking. Intuitively, we think messaging overhead would affect the performance significantly. Several benchmarks generate huge amount of inter-kernel messages as shown in Table 4.3, for example MG and IS. However, from the experimental results the cost of messaging is not a significant factor. Especially, from the experimental results which demonstrate that message passing overhead can be dramatically reduced by utilizing shared memory that has faster inter-connection speed (**SHM**). We also notice that, the results of **SHM** running on 3 different memory model has similar results, because **SHM** always

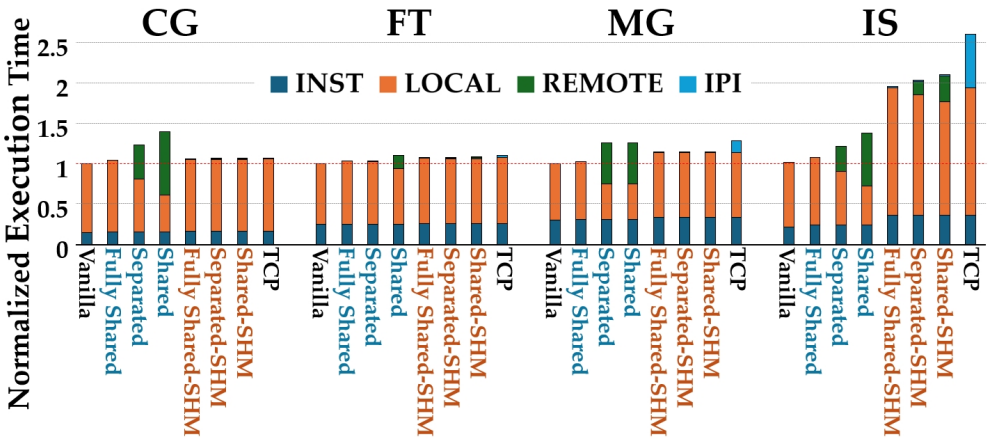


Figure 4.9: NPB benchmark results

replicate the page, the remote memory access overhead is minimum. Thus we consider only use the Popcorn Linux Messaging over Shared Memory(**SHM**) with **Shared** memory model show in our following experiments, if with no further notice, because such memory model is more promising in the near future. Since Stramash Linux also benefit from the faster inter-connection speed, therefore, we consider only use (**SHM**) as our only baseline in following experiments.

Read vs Write Intensive Behaviour. An interesting case is observed where Stramash Linux with **Shared** and **Separated** memory models exhibits weak performance, especially in the CG benchmark, whereas in the IS benchmark, it performs well in all cases. We found that the CG (Conjugate Gradient) benchmark is primarily read-intensive. It focuses on solving large, sparse linear systems using the conjugate gradient method. This involves numerous sparse matrix-vector multiplications, 98.34% of memory instructions are load instructions [10]. Instead, the IS (Integer Sort) benchmark is more write-intensive. It tests the performance of integer sorting algorithms, which would modify the sequence of keys during the procedure stage [19].

4.9.2.2 Cross-ISA migration: Cache Size Sensibility

In previous experiments, each QEMU instances has 4MB of L3 cache. To further compare Stramash Linux and Popcorn Linux, we increased each QEMU instance's total L3 cache size to 32 MB, similar to recently released multi-core processors [9]. The results for CG and IS are shown in Fig 4.10. For CG, which is predominantly read-intensive with fewer writes (and thus fewer invalidations), a larger L3 cache reduces the cache miss rate and overall memory accesses, significantly reduce execution time for Stramash Linux with **Shared** and **Separated** memory models. With a smaller cache, when a read operation loads a new cache line into a full cache, an eviction occurs even without invalidation. In Stramash-Linux, this could involve loading data from remote memory. In contrast, Popcorn Linux (**SHM**) always replicates the page, and the load is always from local memory, so its performance is less affected by cache size changes. As a result, the performance of Popcorn-Linux (**SHM**) in the CG benchmark remains stable regardless of cache size. With a larger cache reducing remote memory accesses,

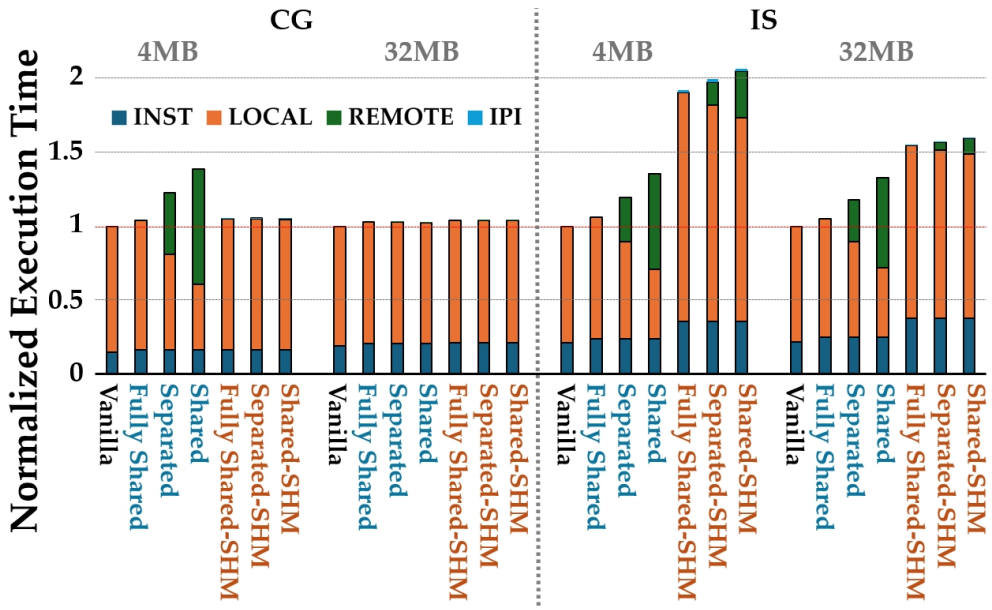


Figure 4.10: IS vs CG, different Cache size result

Table 4.3: Message Count During Migration and Duplicate Page Count During Runtime Migration

	Message Count			Duplicated Pages		
	Popcorn	**Stramash**	**Reduced Rate**	**Popcorn**	**Stramash**	**Reduced Rate**
IS	207124	22	99.98%	16918	7	99.96%
CG	16074	34	99.78%	5603	7	99.88%
MG	287672	6	99.99%	110275	9	99.99%
FT	164702	326	99.80%	98787	16459	83.34%

Stramash-Linux experiences a significant performance improvement. Its slowdown compared to **SHM** decreases from 34% with a 4MB L3 cache to under 1% with a 32MB L3 cache.

For the IS workload, which is write-intensive, the L3 cache faces a high invalidation rate even with an increased cache size, keeping the high cache miss rate. However, in Popcorn-Linux (**SHM**), a larger cache size reduces cache evictions due to the LRU policy. This leads to fewer write-backs as multiple writes can accumulate in the cache before being written back to memory. Each write-back to replicated pages can trigger the DSM consistency policy if a remote node is reading those pages, adding overhead. Therefore, lowering DSM page replication and reducing write-backs improves the performance of Popcorn-Linux (**SHM**) on the IS benchmark. Meanwhile, because the IS workload's cache miss rate remains high, Stramash-Linux continues to access remote memory frequently, resulting in relatively stable performance despite the increased cache size. Thus, Stramash-Linux improvement over **SHM** decreases from $2.1\times$ to $1.6\times$ with the larger L3.

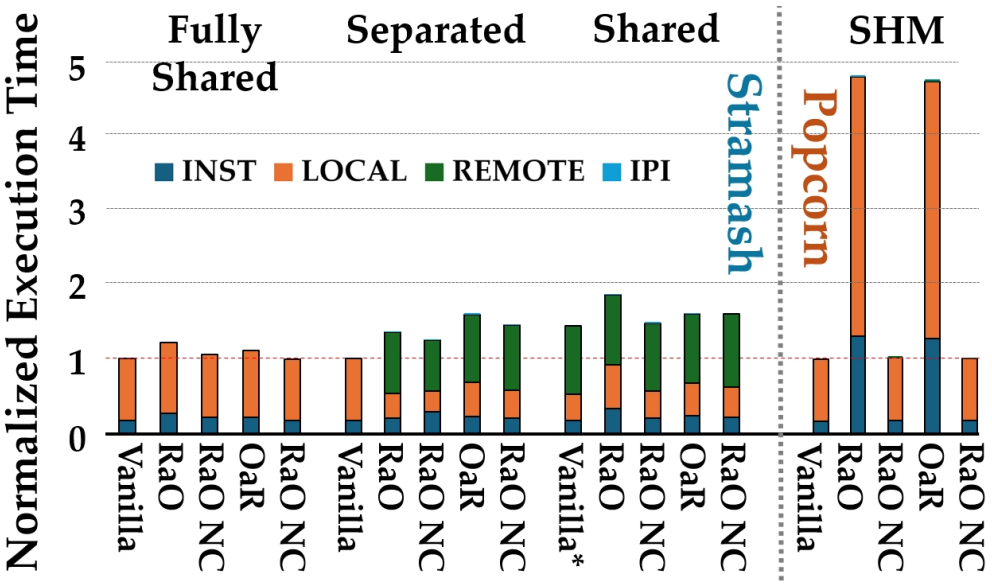


Figure 4.11: Memory Access Analysis (RaO: Remote Access Origin, OaR: Origin Access Remote, NC: No Cold access **Vanilla***: Vanilla experiment runs on `Shared` memory model, **SHM** performs same on all three memory model

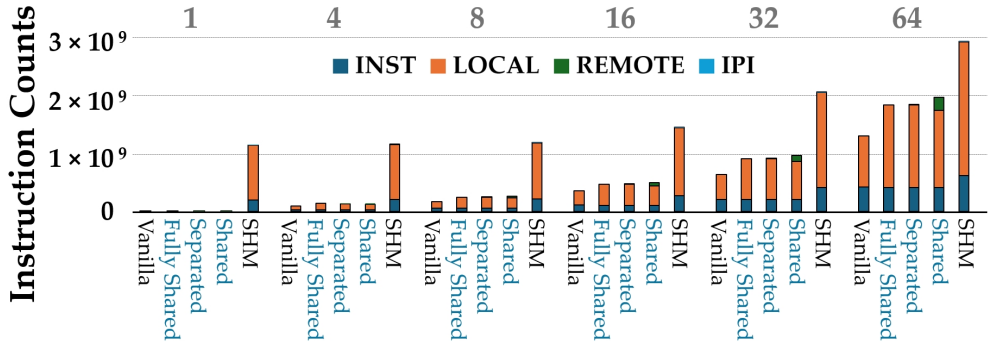


Figure 4.12: Page Access with cacheline granularity, from 1 cacheline size (64 Bytes) to 64 Cacheline size (4096 Byte), **SHM** performs same on all three memory model

4.9.2.3 Cross-ISA migration: Page duplication.

Multi-kernel OS uses DSM to maintain memory consistency. Which replicates pages and necessitates messaging to inform the shared page owner of any updates, thus, software based memory consistency policy – require extra memory capacity. While the Copy-on-Write (COW) policy may mitigate some messaging, significant overheads persist. Stramash-Linux eliminates the need for page replication and ensures that updates are immediately visible to both kernels. Table 4.3 shows the count of duplicated pages during the runtime thread migration. In the current version of Stramash, duplicated pages still exist because it only allows remote kernel allocation at the PTE level. If the upper layer of the page table is missing, the original kernel will handle the fault to reduce complexity.

4.9.2.4 Microbenchmarks: Memory Access Cost.

We developed a memory-bound microbenchmark to investigate the memory access overhead when the DSM protocol is enabled to maintain page coherence. The results are shown in Fig 4.11 (lower is better). In this study, we highlight the performance impact of cross-ISA memory access. We allocate 10 MB of data in either the local or remote kernel instance and conduct sequential memory access operations on the allocated memory. We measured five different memory access activities: The origin kernel accesses the origin kernel's memory, serving as our baseline (Vanilla); the remote kernel accesses the origin kernel's memory (Remote access Origin); remote accesses origin, but the remote kernel has previously accessed the memory and thus has the latest version of the memory content (Remote access Origin No Cold). Origin access Remote case and Origin access Remote No Cold case are similar to the previous cases except with opposite access directions. All execution time results have been normalised to Vanilla.

Results. We can see that Popcorn-Linux Messaging over Shared Memory(**SHM**) has less than 1% of REMOTE overhead in the breakdown, the performance is the same when running on other 2 hardware model, **Separated** and **Fully Shared**. Therefore, Stramash-Linux with the same **Shared** memory model outperforms **SHM** by up to $2.5\times$, and up to $4.5\times$ in the **Fully Shared** memory model.

The most interesting part is the No Cold case. In Popcorn-Linux, the first memory access requires the DSM protocol to duplicate the page and keep the local page content up-to-date. However, during the continued warm access (read or write), the local pages are already updated, resulting in no DSM overheads, and performance is close to the vanilla case since all memory access is local. In contrast, Stramash-Linux with the **Shared** or **Separated** memory model performs weaker because of remote memory access overheads, since there is no page replication, it needs load from the memory again if the previous accessed data have been evicted from the cache, and in the worse case the data is loaded from the remote memory with extra overhead. Thus, application with access pattern that exclusively and frequently access shared data will be benefit from the page replication. However, we observe that Stramash-Linux can significantly reduce the overhead of the traditional multi-kernel DSM protocol and achieve performance comparable to the SMP case when deployed on a **Fully Shared** memory model.

4.9.2.5 Microbenchmarks: Software vs Hardware Consistency.

Software Distributed Shared Memory (DSM) is notorious for its overheads, especially as systems scale up [61]. Overheads stem from the expensive maintenance of data consistency when data, often in page size, is replicated across multiple nodes, which requires inter-node communication that is expensive. However, none studied its performance when nodes are tightly connected like in **Shared** model. In contrast, Hardware-based Cache Coherence, such as that supported by CXL 3.0, enables data transfers at cacheline granularity. To quantify the performance differences between software and hardware consistency, we conducted experiments by accessing data ranging from a single cacheline (64 bytes) up to 64 cachelines (4096 bytes) – one page.

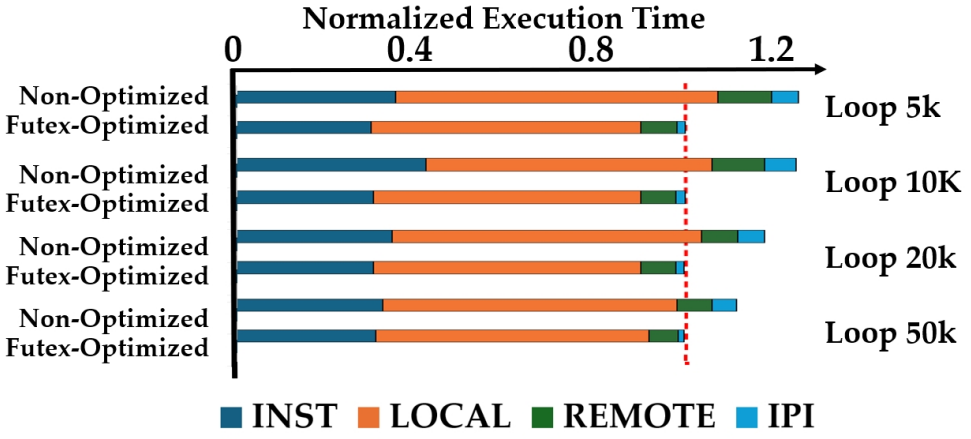


Figure 4.13: Futex experiment results

Results. As shown in Fig 4.12, when accessing just one cacheline, DSM incurs an overhead exceeding 300× compared to hardware coherence approaches, due to the unnecessary replication of entire pages. Even when accessing all data within a page, hardware coherence can still achieve approximately 2× faster performance than DSM. These results underscore the inefficiencies of software DSM for applications with scattered fine-grained data access patterns, and its potential benefits when sequential access is needed.

Lesson Learn: Software Coherent vs Hardware Coherent.

Software Coherent: More page replication require extra memory capacity, total cache and memory ops are higher; Maintain page consistency is expensive,pages need copy again if it is invalid; Exclusively access show good performance, as there is only local memory access.

Hardware Coherent: Less page replication, less memory capacity needs; Maintain page consistency is cheap, page invalidation only require reload to the cache; Frequently access show bad performance when the data is in remote.

Key take away: Applications that frequently and exclusively access remote data can benefit from data replication. Depending on an application's memory access patterns, there is a trade-off between different approaches to memory coherent. We propose a hybrid solution that combines both software and hardware memory coherent methods. As systems scale, the CXL snooping filter tables can become enormous, especially when managing a large range of cache-coherent shared memory among multiple nodes. In such cases, we believe that Software Coherent method (DSM) can efficiently manage less frequently modified pages, alleviating the overhead associated with Hardware Coherent method.

4.9.2.6 Microbenchmarks: Futex Lock.

In Popcorn-Linux, all Futex instances are created and managed by the origin kernel, while Stramash-Linux allows the remote kernel to handle the Futex operation itself. We set up an experiment to demonstrate the performance improvement of Stramash-Linux in handling Futex operations – with and without Futex Optimization, to elim-

Table 4.4: Memory allocator overhead of performing offline and online operations with different memory slice sizes

	Qemu-x86		Qemu-Arm	
Num of Pages	**Offline**	**Online**	**Offline**	**Online**
2^{15}	12.5ms	5.8ms	4.8ms	5.8ms
2^{16}	27.7ms	10.9ms	16.1ms	12.3ms
2^{17}	38.6ms	14.3ms	14.3ms	16.7ms
2^{18}	66.6ms	22.6ms	21.1ms	28.8ms
2^{19}	129.4ms	36.5ms	36.4ms	42.6ms
2^{20}	246.3ms	68.1ms	64.4ms	80.9ms

inate the impact of all other factors. The Futex microbenchmark: the origin kernel continuously locks the Futex while the remote kernel continuously unlocks the same Futex, performing a simple addition in each loop. A higher loop count indicates more Futex operations.

Fig 4.13 shows the execution time results of the Futex experiment, comparing the Stramash Futex-optimized case to the regular case. The execution time result has been normalized to the Stramash Futex-optimized case. With optimization enabled, the plain instructions are reduced, leading to decreased memory access overhead. Additionally, we see a reduction in message cost. In the Futex-Optimization case, only one cross-ISA IPI is needed to wake up the waiting thread, whereas the original solution requires a full Futex management protocol, including multiple requests and response messages to handle each Futex operation. For example, in the loop 50k case, the cross-ISA messages are reduced from 102,363 to 25,723.

4.9.2.7 Global Memory Allocator Overheads.

Dynamic physical memory allocation between kernel instances offers the potential to improve overall platform memory utilization compared to static partitioning. The benefits of improved memory utilization with global memory allocators have already been proven. Here, we present the overheads introduced by our allocator, including the time required for offlining and onlining memory slices. We set up Stramash-QEMU with 4GB of memory dynamically shared between two kernels. Each memory slice in our setup contains 2^{16} pages (256MB), resulting in a total of 16 slices. We recorded the average time to offline or online a slice on both the Arm and x86 sides. Table 4.4 illustrates the overheads introduced by our allocator, measured in milliseconds, primarily due to the page isolation process.

4.9.2.8 Network-serving Application.

In this experiment, we are using Redis-server [150] as an example of a network-serving application. We modified the Redis-server to migrate between ISA CPUs. Because we cannot migrate a process/thread that reads/writes to a socket – a Popcorn-Linux limitation, our modified Redis-server migrates to the remote kernel during the processing of the `time_event`. We do not enable the Stramash-QEMU Cache plugin because the simulation has a different time model compared with real-time, plus the simula-

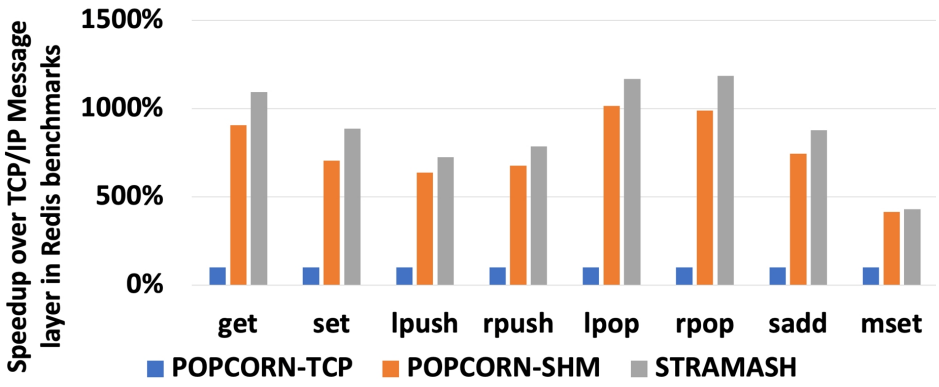


Figure 4.14: Redis speedup

tion is slow enough to make connections timeout. On the host machine, the `TCP/IP` based message layer is connected through a Linux network bridge, which connects the tap device of the QEMU. The Redis benchmark runs with 10K requests with fixed payload sizes at 1024 bytes. Due to the different timebase of client and server, we measure processing time for each round of requests inside our modified Redis-server.The normalised results are shown in Fig 4.14, with the `TCP/IP`-based message layer (`POPCORN-TCP`) set as the baseline, higher is better. The experiment shows that `SHM`-based message layer (`POPCORN-SHM`) can gain around $4 - 10\times$ speedup compared to the baseline. With Stramash-Linux enabled, the speedup can be even greater, up to $12\times$.

4.10 Discussion

We have explored three different fused-kernel deployment models from an OS perspective: **Separated**, **Shared**, and **Fully Shared** (see Figure 4.3). The differences among these models relate primarily to how each kernel instance views system memory and how they synchronize or communicate with one another.

Separated Model and Shared Model. In a **Separated** model, each kernel instance runs its own OS image and manages a distinct, private memory space. This design aligns with common multi-kernel approaches, where communication between kernels is achieved through explicit mechanisms (message passing through network, TCP/IP or RDMA). Consequently, from an OS-level perspective, the kernels remain isolated: any shared state must be carefully marshaled through higher-level interfaces. With CXL, a Separated model can support a fused OS in the sense that coherence can be provided by hardware. The OS, however, must enforce boundaries on each kernel's "exclusive memory" region to prevent unauthorized direct access by other kernels. Only selected shared address ranges are made accessible (if at all) for remote memory operations, preserving strong isolation but still allowing where needed.

The **Shared** model introduces partial memory sharing across multiple kernels. Here, the OS coordinates a "shared pool" accessible to all kernels with cache coherence, which is supported by CXL. Each kernel retains its own exclusive memory, invisible to the others. Access to or sharing of that exclusive memory is still possible

through regulated OS-level synchronization (e.g., semaphores, locks, message passing).

In practical terms, both the Separated and Shared models can be implemented under the upcoming CXL 3.0 specification.

Fully Shared Model. In a **Fully Shared** model, all heterogeneous kernels collectively manage memory as if they were components of a single unified OS image. While projects such as OpenPiton or BYOC [55, 56] address these questions at the microarchitecture level – designing cache-coherent frameworks that can flexibly attach disparate heterogeneous cores, which provide a standardized private-cache layer (often the L2) and an LLC shared by all cores. The L1 caches remain tightly coupled to each heterogeneous core's pipeline and instruction frontend, so they typically remain hidden from the coherence framework. By separating L1 from the rest of the cache hierarchy, the underlying system can implement a coherent memory model without requiring each core's L1 to be re-engineered for protocol compatibility. Overall, the Fully Shared model aspires to present a single-system illusion across multiple kernels and cores. Yet, achieving that vision necessitates a robust underlying hardware coherence framework.

4.11 Conclusion

With research in academia showing the potential benefits of heterogeneous-ISA cache coherent platforms, and emerging cache coherency interconnects over peripheral busses to connect heterogeneous-ISA processors on the same platform, it is fundamental to investigate how operating systems software on such platforms should look like.

In this work we propose a new Operating System design targeting such emerging platforms, the fused-kernel OS, which is a multiple-kernel OS that maximally exploits cache coherent shared memory for inter-kernel coordination.

Because real-hardware of our target platforms is not commercially available yet, we built an hardware simulator based on QEMU and Cache-plugin for memory-accurate simulation. We called this Stramash-QEMU. We used Stramash-QEMU to compare the fused-kernel OS implemented by Stramash, versus the state-of-the-art multiple-kernel OS for heterogeneous hardware Popcorn Linux on 3 different hardware models. We discover that Stramash-Linux enables best application performance in most of the cases, but performance improvements depend from an application access pattern and the hardware model. Amongst others results, our simulated CXL 3.0 hardware model shows that using DSM with Popcorn-Linux may results in better application performance than direct accessing remote memory with Stramash-Linux, for at least NPB CG.

We believe Stramash could be used for further research on CC shared-memory platforms, therefore, all code will be released open-source.

Chapter 5

Tiered Memory Management

5.1 Intruduction

In cloud computing environments, providers deploy virtual machines (VMs) across multiple global regions, each comprising several server clusters. Within these clusters, a central scheduling system handles client requests by allocating on-demand VMs across different servers. Each server is equipped with a local management unit responsible for managing the entire lifecycle of the VMs, from creation to termination. A significant challenge in these platforms is the underutilization of resources such as CPU, memory, and network bandwidth. Major cloud providers have highlighted the issue of resource underutilization. Research from Azure [88] indicates that memory allocation ranges between 50% and 70% for most servers. Similarly, a study by Alibaba [169], examining over 10,000 VMs during a one-week period, revealed that about 25.3% of the VMs' memory was consistently unused.

Resource Underutilization. To address resource underutilization issues in cloud platforms, several solutions have been proposed and deployed, such as dynamic resource harvesting and overcommitment techniques for all different types of hardware resources, such as CPU, memory, and I/O resources. For instance, fixed-size Spot VMs [41] have been used to overcommit CPU resources. Ambati et al. [42] have suggested a more efficient approach with the introduction of Harvest VMs. These VMs receive a fixed amount of memory but can access any spare cores available on the same server, which reduces VM eviction rates and lowers costs in comparison to traditional Spot VMs. Burstable VMs [52, 53] apply a similar concept but adjust the number of physical cores assigned to a VM based on a credit system while keeping the memory allocation constant. Elastic VMs [170] extend this idea by not only reclaiming unallocated cores but also those that are temporarily inactive, although their memory allocation remains unchanged.

However, these currently deployed works in the cloud focused on utilizing the CPU cores while keeping memory size fixed. Memory is expensive and can cost roughly 50% of server costs today[21], so underutilizing the memory resource represents a significant loss for the cloud provider. A recent innovation from Azure is the Memory Harvest VM [88], which specifically targets memory resources. This solution allows VMs to dynamically adjust their memory sizes based on the availability of unallocated memory and to quickly release acquired memory when a high-priority, on-demand

VM arrives. This adaptive strategy aims to significantly enhance memory resource utilization efficiency in cloud environments. However, with the introduction of new hardware technologies into the cloud, previous approaches might not fully exploit the benefits that this new hardware can bring.

Memory Pooling. The emerging Compute Express Link (CXL) standard [78] supports memory pooling — which offers additional memory capacity and reduces resource fragmentation by allowing direct access (load/store) to the memory extender through the PCIe bus, and such extended memory can be shared among multiple servers. Memory pooling can greatly improve resource allocation flexibility and reduce memory stranding in data centers. However, pooling memory often suffers from relatively higher access latency compared to traditional directly attached DRAM, which can potentially affect application performance.

Studies like *Pond* [119], which analyze the performance impact of memory pools supported by CXL, indicate that allocating latency-tolerant applications to the memory pool does not adversely affect performance. For example, they found that 26% of the 158 workloads they have investigated are experiencing less than 1% slowdown when fully deployed at the CXL memory pool, and an additional 17% of workloads see less than 5% slowdowns. Moreover, by exploiting the extra memory capacity from the CXL pooling memory, this leads to a 7% reduction in overall DRAM requirements, translating into a 3.5% cost reduction for cloud servers. However, not all applications are latency-tolerant. The study from *Pond* also found that 21% of workloads experienced more than a 25% slowdown when deployed on a CXL-based remote memory pool. For these latency-sensitive applications, current research focuses on efficient partial deployment – allocating memory from both local and remote memory pools. This approach detects application memory access patterns and adopts page replacement strategies that promote frequently accessed (”hot”) pages to local DRAM while moving less frequently accessed (”cold”) pages to the remote memory pool. By properly allocating pages into the correct tier, the performance impact can be minimized while still enjoying the extra capacity from the memory pool. The key idea behind this strategy is to keep frequently accessed data closer to the computing resources, which requires monitoring memory accesses to determine which data should remain local.

Profiling in Virtualization. To obtain accurate memory access patterns of applications, these strategies typically rely on page-level profiling, such as scanning access bits or page table entry (PTE) poisoning [36, 101, 153], or on hardware-accelerated instruction sampling techniques like Intel PEBS [85, 116, 148] and AMD IBS [43]. While most of the work targets bare-metal applications, very few address virtualization environments. In virtualization, page-level profiling is resource-intensive, consuming significant host CPU cycles and often leading to suboptimal decisions, because it needs extra translation to translate guest virtual addresses to the host physical address due to the address space isolation from the virtualization. While the hardware sampling currently does not support profiling the guest operation[153]. These challenges make software-based memory replacement techniques less appealing for deployment in current cloud virtualization environments.

Problems. To fully exploit the extra memory capacity provided by CXL, current solutions focus on application profiling, fixed partial allocation of memory between fast local DRAM and slower remote memory pools, and runtime profiling of memory access patterns to dynamically shuffle hot and cold pages between local and remote

memory. However, fixed partial allocations might not fully utilize memory resources because the available local DRAM capacity can be dynamically adjusted in the cloud. Moreover, page-level shuffling relies on memory access pattern profiling, which may not work effectively in virtualization environments. After memory shuffling, the memory topology can become fragmented, making NUMA-spanning issues more likely to occur.

Additionally, much of the existing research focuses on application-level rather than VM-level optimization, often prioritizing a single target workload performance without adequately considering the complexities of the cloud environment. This can result in aggressive strategies that may disrupt the performance of other VMs. Unlike studies targeting bare-metal scenarios [116, 130, 148], cloud environments involve multi-tenant settings where multiple VMs compete for local memory resources. This competition makes it impractical to assess page hotness across different VMs effectively. Current research predominantly addresses single-tenant use cases, focusing mainly on optimizing individual VMs to maximize workload performance by leveraging more remote memory and minimizing local memory usage. This approach often overlooks the memory contention caused by other tenants in a multi-tenant cloud environment. For instance, when multiple VMs are competing – all attempting to promote their hot data into the limited local DRAM, it becomes exceedingly difficult for the host hypervisor to make accurate page-level decisions that fairly allocate resources among all different tenants. This contention can lead to performance degradation and unfair resource distribution, highlighting the need for strategies that consider the multi-tenant nature of cloud infrastructures.

Libra. *Therefore, we argue that in virtualization environments, having the host profile the guest's memory access patterns and then performing page-level optimization by migrating page between local and pooled memory is not an effective strategy.* Instead, the role of the hypervisor should be redefined to coordinate resource allocation at the VM level. Rather than focusing on fine-grained page-level management, the hypervisor should concentrate on VM-level optimization.

We introduce **Libra**, a system that monitors the overall memory access pattern of each VM to prioritize them and adjust their memory allocations between local and pooled memory accordingly. Libra enables dynamic re-proportioning of VM memory allocations, allowing the ratio of local to remote memory to adapt according to current resource availability as the capacities of different memory tiers change over time. When a VM's local-to-remote memory proportion changes, Libra exposes these topology changes to the guest VM, making it aware of the adjustments and enabling it to manage its internal memory allocation based on the updated memory topology. This strategy can be easily implemented and integrated within existing cloud infrastructures while maintaining or enhancing performance. Our approach aims to balance resource optimization with operational simplicity, making it a viable and practical solution for contemporary cloud platforms.

Contributions. We make the following key contributions:

- *First*, we showcase the variability type of virtualized workload that is not feasible for host profiling. Moreover, we analyze why previous works failed in providing solutions that can be deployed by Cloud providers.

- *Second*, we introduce Libra, it is designed around four key ideas: Minimal changes to guest VM software; Ease of management; VM level profiling; and guarantee performance fairness.
- *Third*, we implemented and evaluated an initial prototype of Libra dynamical memory topology setup. Showing that Libra can eliminate the randomness of memory allocation and improve memory utilization.

5.2 Background and Motivation

5.2.1 Hypervisor memory management

In the public cloud, providers typically offer infrastructure where cores and memory are exclusively allocated to a tenant's VM. This strict allocation is crucial to prevent out-of-memory (OOM) issues during runtime, which could lead to service disruptions—unacceptable in environments where reliability is paramount. To avoid such scenarios, hypervisors employ a strategy of statically preallocating—or "pinning"—the entire address space of a VM into the host's memory. This approach guarantees that each VM has consistent and reliable access to the resources allocated to it, aligning with the fundamental principles of Infrastructure as a Service (IaaS). This ensures that VMs operate within their resource limits, providing stability and predictability for both cloud providers and their customers.

5.2.2 Resource Limitation

Cloud platforms schedule VMs based on a vector of resource requirements, such as CPU and memory. A resource is considered stranded when it is technically available for rental but practically unusable because another necessary resource has been exhausted. A typical scenario for memory stranding occurs when all server cores have been rented out, leaving excess memory that cannot be utilized. Conversely, memory spilling happens when there are unused cores, but a lack of memory prevents the allocation of a VM to that server.

Due to constraints inherent in Cloud VMs' schedulers [95, 118], CPU overcommitting practices [94] and tenants' tendencies to request more VM memory than needed to buffer against potential usage peaks [169], servers often struggle to closely match the resource demands of incoming VMs. This overprovisioning complicates the tight packing of VMs, especially when the DRAM-to-core ratio of VM arrivals does not align with the available server resources. Although [118] suggests that memory stranding is now not an issue in the Cloud, they indicate that with the future deployment of CXL-based pooled memory systems, the likelihood of memory stranding may increase.

5.2.3 Infrastructure challenge

Unused memory on one server cannot be utilized by others, leading to memory stranding. Implementing tiered memory and a rack-scale memory pool system [102] can improve VM placement effectiveness. However, Pond [119] showed that 21% of the

tested workloads experienced performance slowdowns of more than 25% when allocated 100% remote memory, which conflicts with QoS requirements.

Previous research [153] has focused on dynamically shuffling hot pages with minimal local memory to maximize performance for VMs with mixed memory allocations. Yet, this solution is not yet ready for deployment in real Cloud environments. The issue is not solely the cost of host profiling at the guest page level but also a lack of consideration for the infrastructure differences in the Cloud. For instance, to maintain a low profiling cost, the existing solution conducts profiling every minute. However, more than 15% of VMs are allocated for durations of 10 minutes or less [95], rendering such frequent page profiling ineffective. Additionally, memory utilization within each server in a compute cluster fluctuates over time. Cloud servers are not always fully loaded, and issues of memory stranding or spilling typically occur only during peak stages. For example, according to a study by Azure [95], the number of VM requests varies throughout the week, dropping to as low as 10k/h at night and reaching up to 40k/h during peak periods. Therefore, how to dynamically adapt memory optimization policies remains an unresolved challenge in Cloud computing.

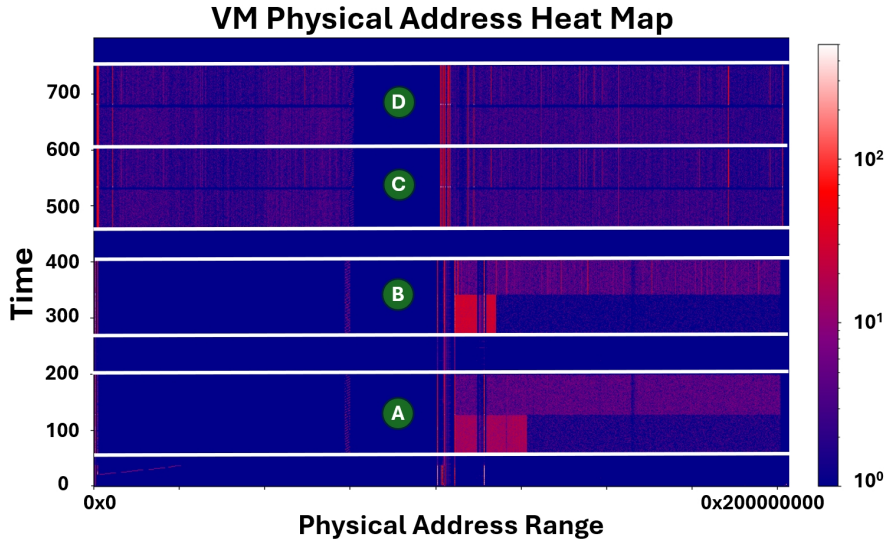


Figure 5.1: *Memory Sparse:* (A), (B) data is inserted into an empty redis-server in a fresh booted VM, (C),(D) data is reloaded from the disk storage. (A)/(C): 20% Hotspot access + Uniform Access; (B)/(D): 10% Hotspot access + Uniform Access

5.2.4 Workloads profiling challenge

It is a challenge for the host to profile the virtualized workload in the Cloud. We have profiled different types of workloads; the results are shown in Fig 5.1 and Fig 5.2. More details of the experiment are explained at Sec 5.6.1. Based on our observation, we can conclude that not all virtualized workloads are worth profiling. We have identified at least three types of workloads where profiling can be problematic:

- **Sparse Memory Workloads:** These workloads span the entire memory sparsely. Profiling these requires monitoring the whole VM memory. For example, in key-value stores, accessing only 20% of the keys doesn't mean only 20% of the physical

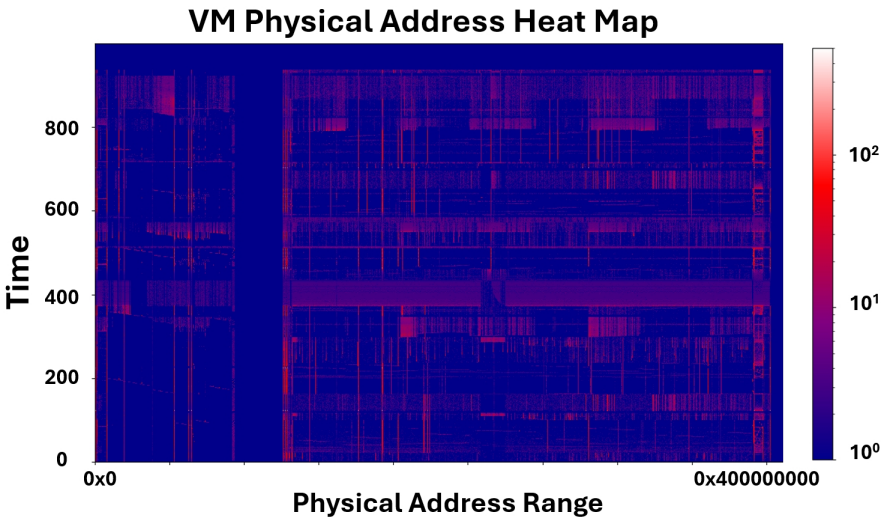


Figure 5.2: *Different workloads:* YCSB mix of Zipfian, Latest, Hotspot access pattern to a redis-server with different read, insert and update rate

memory is touched. The workload will likely touch most of the memory, necessitating a full physical address space profiling.

- **Dynamic Hotspot Workloads:** These workloads change their hotspots at runtime, requiring frequent updates to the memory profile. This frequent profiling is costly. For instance, newly inserted data is likely to be accessed again, causing the workload's hotspot to shift continuously. Profiling would need to constantly adapt, leading to high overhead.
- **Short-Lived Workloads:** These workloads finish fast, rendering memory profiling ineffective. For example, function executions in serverless computing often take seconds [155]. In such cases, the profiling process and state updates need to be frequent, or they become irrelevant.

Given these challenges, our solution focuses on profiling at the VM level to detect activity patterns and prioritize VMs. This approach avoids the need for constant, granular memory profiling and is more feasible for real Cloud deployments.

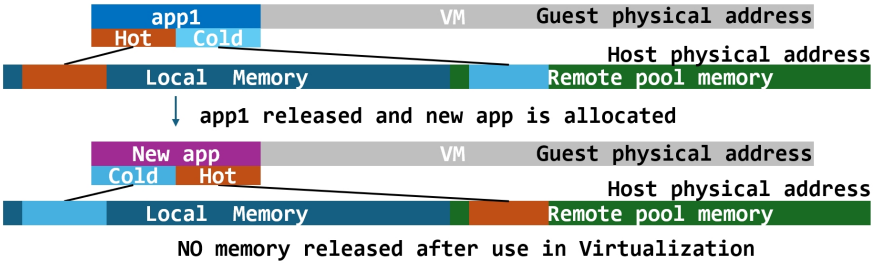


Figure 5.3: *Host to Guest memory mapping is remaining after Guest application released*

5.2.5 Virtualization challenge

The software-based tiered-memory page replacement policy, which replaces the hot and cold pages between fast and slow memory, is not optimal in a virtualized environment. Because the guest VM page is backed up by the host page, when the host migrates the guest page, the VM is unaware of this page relocation. Unlike applications that release memory after use, VMs retain pages until eviction. As shown in Fig 5.3, if a VM starts a new application, the previous guest-to-host memory mapping remains, and the newly allocated application might use the remote memory even with enough local memory available; in other words, the page migration will cause the NUMA-spanning issue in the VM. The page replacement policy needs to spend extra cycles to profile the hot page from the newly allocated pages and migrate them back from the remote memory to retain the performance. Therefore, the host's transparent migration of VM pages results in physical memory being fragmented in the guest.

Exposing NUMA topology. Exposing the tiered-memory topology to the guest is not able to solve the issue. First, it will sacrifice the flexibility of dynamically adjusting the local and remote memory. In other words, with the tiered-memory topology exposed, it is more like memory ballooning; the guest needs an extra memory zone to manage the pages. Second, it needs hot-plug-like technology to support dynamical memory resizing, especially when releasing the memory. The VM needs to isolate a continuous range of physical addresses, and an unmovable page could fail the operation[102]. Previous works like XPV [67] and HeterOS [110] both address such problems by exposing the memory topology to the guest VM. Specifically, XPV targets the NUMA-spanning problem; it ensures the reclaimed page from another NUMA node is placed correctly into the correct page allocator's list. Because reclaimed pages from another NUMA node are determined, the guest knows which node the reclaimed page belongs to. However, with the page replacement policy, the guest page location is not deterministic at runtime. Another work is HeterOS, which introduces a special NUMA node inside the guest. This special NUMA node only contains the local memory, and the guest uses the special API to allocate the local memory page exclusively. However, due to the fast local memory being assigned exclusively, this could still cause resource wasting in the Cloud.

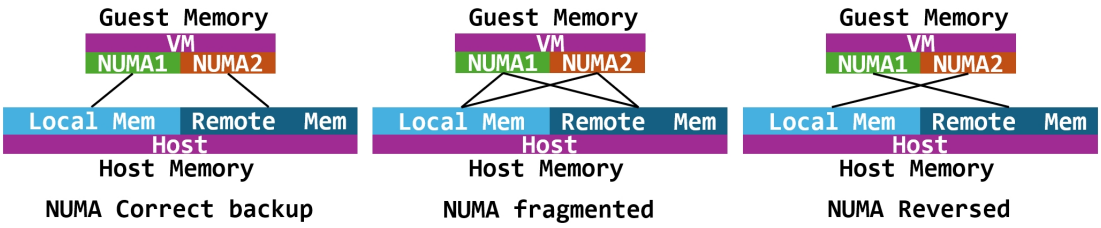


Figure 5.4: *Guest to Host memory mapping issue*

Exposing the NUMA topology to the guest has another problem shown in Fig 5.4. With the page replacement policy, in the worst-case scenario, the guest local and remote NUMA node memory mapping to host memory is completely reversed. The guest local NUMA node memory is backed up by the host remote memory, and vice versa.

5.3 Design principles

Libra is based on the following ***design principles***:

- **Minimal Changes to Guest VM Software**
  – Community friendly: Minimal modifications to guest VMs
- **VM-Level Profiling and Reproportion:**
  – VM Priority: Profiling is conducted at the VM level rather than at finer granularities like the page level.
  – Re-proportion: Change VM's local-remote memory proportion based on demand
- **Performance**
  – Transparent-NUMA: Host update the guest memory topology, Guest is aware of memory topology change

5.4 Libra Design

Libra extends modern hypervisor memory subsystem in order to:

- **Accurate Prioritization** *Accurate monitoring of VM status:* Based on the current server's local memory availability, Libra assigns varying local-remote memory priorities to the VM. The host VM scheduler then makes the final decision on which VM should receive more local memory, thereby optimizing performance based on current demands.
- **Dynamical NUMA topology** *Dynamically adjust guest NUMA topology address:* To minimize the impact of NUMA misplacement issues, Libra change the guest memory mapping to different memory-tier and update the latest topology info to the guest; and guest dynamically change its NUMA topology to efficiently manage the memory.
- **Guest-led performance optimization** *Libra's differentiated policies:* Libra rethink the role of host and guest, the guest take care of the application runtime optimization by leveraging the page access profiling tool to identify the temperature of the page, and according to the latest NUMA topology to place the data of different workloads correctly.

5.4.1 VM profiling

Rather than focusing on profiling hot and cold pages, Libra focuses on identifying "hot" and "cold" VMs. Previous methodologies, such as those utilizing Intel PEBS for hot memory profiling, are not well-suited for virtualization profiling since the host cannot accurately retrieve the correct virtual address from it. However, Libra still uses PEBS L3 cache miss counts, *MEM_LOAD_RETIRED.L3_MISS* as a metric to determine if a VM is actively engaging in memory-intensive activities. Alternatively, the Damon [144] page profiling tool could also be used. Beyond this metric, we also incorporate I/O interrupts as additional evidence to evaluate the priority level of a VM. This is based on the understanding that VMs generating I/O requests are likely to be latency-sensitive. VMs with fewer I/O requests are considered best-effort workloads. This dual metric system allows Libra to more effectively classify and manage VMs based on their operational characteristics and performance requirements.

Libra's VM-level profiling primarily identifies whether a VM is running "hot" or "cold," in terms of resource usage. In scenarios where all VMs are of the same priority—effectively all "hot," Libra faces a challenge in prioritizing one VM over another. However, this raises an important consideration: is prioritization necessary in such a scenario? When all VMs are uniformly demanding, the likelihood is high that they all require more resources. In such cases, attempting to prioritize one VM could lead to suboptimal resource distribution and increased contention. Therefore, Libra can recognize when the server is at a high peak usage stage. During these times, Libra opts not to perform any page migration. This decision is strategic to minimize the impact of memory contention between VMs. By avoiding page migrations during peak times, Libra helps maintain stable performance across all VMs.

5.4.2 Memory scheduling

During each Libra's "memory scheduling" period, the host memory scheduler scans all VM statuses in the server. It assigns priorities to each VM based on their current memory or IO activity. The local and remote memory proportion allocation strategy currently employs a FIFO policy, prioritizing VMs from highest to lowest.

Libra categorizes VMs into 3 distinct priority levels:

- **VMs with both I/O and memory access** - These are considered highest priority as they are likely engaging in latency-sensitive tasks that require rapid access to resources to maintain QoS.
- **VMs with only memory access** - This category captures VMs that might less sensitive to latency than the first group but still require prompt access to memory.
- **All other VMs** - This category is treated with the lowest priority, as these VMs are least likely to be affected by slower memory access times.

This prioritization ensures that latency-sensitive workloads receive the memory resources they need promptly, while VMs without active memory demands can yield their local memory resources back to the system. This approach helps in efficiently managing memory resources, particularly during peak times when most VMs are experiencing high demand.

A potential downside of this strategy is that our local DRAM allocation might be sparser compared to systems that concentrate hot data into local DRAM. This could mean a trade-off between maximizing resource utilization and minimizing latency for high-priority tasks. Despite this, the approach is designed to balance overall system performance with fairness, ensuring that all VMs receive adequate resources according to their operational needs and priorities.

5.4.3 VM memory re-proportion

Host-Leading Approach. Such as VTMM [153] the tiering decisions are made within the hypervisor. Host-leading page migration relies on profiling memory access patterns to align data placement with workload demands. However, obtaining a precise profiling heatmap from the host side can be a challenge, as we discussed before. Furthermore, scalability concerns arise from hypervisor-level profiling overhead. Tracking per-page "temperature" across thousands of VMs strains host resources, a bottleneck

that grows prohibitive at scale – shifting all guests' workloads profiling and optimizations to the host is impractical.

Guest-Leading Approach. Given the limitations of host-leading approaches, a guest-leading approach appears theoretically superior as VMs have direct visibility into application behavior and memory access patterns. Such an approach would allow VMs to make proper decisions about memory tier placement based on a deep understanding of their workloads.

However, several practical constraints prevent implementing a sole guest-leading solution. First, VMs lack visibility into the underlying physical memory topology and host-level resource constraints. This information gap can lead to misaligned optimizations. Second, in multi-tenant environments, uncoordinated memory claims from multiple VMs could lead to resource contention and potential system instability. Additionally, current hardware architectures don't provide mechanisms for guests to directly control memory tier placement while maintaining proper isolation between VMs. These limitations mean that while guests are best positioned to understand their memory needs, they cannot effectively manage tiered memory allocation entirely on their own.

Libra. Rethink the role of the guest and host; guests should self-adjust to the memory footprint change and self-profile to optimize the hot/cold page placement. While hosts should only dynamically adjust the VMs' memory proportion based on requests from a global memory scheduler, instead of looking into each VM's workloads and manipulating the guest memory address space directly.

5.5 Implementation

We developed the initial prototype of Libra using the Linux kernel version 6.3. The prototype features an extension of the host hypervisor, with over 3000 LoC added to the Linux kernel; note that currently the project is under active development. These additions improve the monitoring capabilities for current VMs and overall server resources, thereby facilitating more effective resource allocation decisions. Libra will be released open-source upon publication in the academic conference.

5.5.1 Libra's Dynamic NUMA Topology

Limitations of Traditional Hotplug. In Libra, we enable dynamic NUMA topology reconfiguration within guest environments by leveraging a modified mechanism inspired by the Linux Hotplug subsystem. Although the conventional Hotplug facility is specifically designed to adjust the memory capacity of a system at runtime, it inherently assumes that memory sections are physically added to or removed from the system. Consequently, it does not directly facilitate the redefinition of NUMA boundaries without altering the overall physical memory footprint.

The standard memory Hotplug procedure exhibits several drawbacks that make it unsuitable for our dynamic NUMA-topology adaptation. Because it requires offlining a contiguous range of physical memory that must be free, achieving this state demands

migrating all resident pages from the targeted range, which would cause at least 3 problems:

- 1. Unmovable pages: Some guest pages cannot be migrated due to being locked or in use by critical system processes, which may failed the hot-plug opreations;
- 2. Out-of-Memory (OOM) situations: When attempting to off-plug memory, the VM may encounter OOM conditions if there isn't sufficient free memory to accommodate the data being moved, thus the workloads could be killed during the hot-plug process;
- 3. Resource contention: To prevent the OOM, guest could Hot-plugging new memory before un-plugging old memory; however, it requires temporary extra memory space, which may not be available for the current host; further, the internal migration is inevitable – it still might encounter the Unmovable pages issue.

Therefore, we argue that the fundamental goal of Libra is to dynamically change the guest NUMA topology only – without performing large-scale data movements – rather than to adjust the memory capacity. To address the limitations of Hotplug, we implement the "soft Hotplug" mechanism optimized for topology reconfiguration.

Soft Hotplug. Soft Hotplug introduces a more flexible approach that minimizes page migration and exploits existing kernel primitives. The process unfolds as follows:

- Isolating Memory Ranges: We begin by identifying the memory ranges targeted for NUMA topology reassignment and marking them as isolated. This isolation leverages existing kernel mechanisms to ensure that ongoing allocations will not place new pages into these regions, thereby stabilizing the configuration.
- Handling Allocated Pages In-Place: Unlike traditional Hotplug, which demands that all allocated pages be migrated out of the offlined range, soft Hotplug leaves them intact. Allocated pages remain mapped and tracked by existing page tables. When these pages eventually become free, they are then correctly placed into the newly designated allocator structures, see below.
- Removing Free Pages from the Original Zone: With allocated pages stay "untouched" in place, all free pages within the targeted memory range can be cleanly removed from the original zone's free lists. This step also offlines the memory section in the old zone's context.
- Populating a New Zone: Once freed from the old zone, these memory sections are onlined into the new zone associated with the desired NUMA node. The previously isolated free pages are integrated into the new zone's allocator structures, thereby establishing the revised NUMA topology without extensive data movement.
- Metadata Adjustments for Allocated Pages: Finally, we update the metadata of any still-allocated pages so that, upon their eventual deallocation, they return to

the correct zone (the new, NUMA-adjusted zone) rather than the original one. This ensures that the allocator consistently respects the updated topology over time.

By adopting this soft Hotplug approach, Libra avoids the bulk of runtime overhead and complexity associated with traditional Hotplug-based capacity changes. Physical pages remain properly tracked and managed by the newly assigned zone's allocator, and no disruptive large-scale memory migrations are necessary.

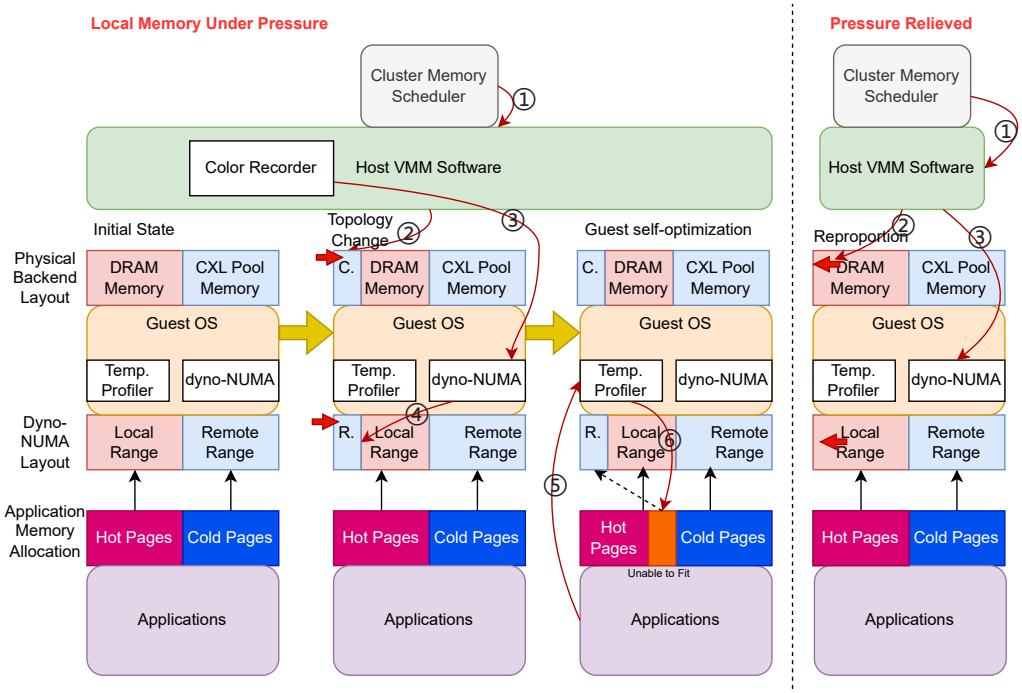


Figure 5.5: SmartVM Architecture

5.5.2 Libra framework.

The SmartVM framework is constructed by following software components:

- **Cluster Memory Scheduler**: Manage memory capacity in cluster globally based on priority and system-wide pressure, adjust the VMs memory proportion(Local-Remote) on demand.
- **Host Coloring Recorder**: On host side, after VM's memory proportion changed, it records the topology of guest-host memory mapping and update to dyno-NUMA inside the guest.
- **dyno-NUMA**: Based on the latest host-guest memory mapping provided by the Host Coloring Recorder, to adjust guest NUMA topology from Guest OS perspective (soft hot-plug).

- **Guest Temperature Profiler**: Leverages page access profling tool, such as DAMON [144], to monitor applications' memory access patterns, identifying page temperature and migrate the page to adapt memory topology change.

Workflow Example. As illustrated in Figure 5.5. Upon a peak in host local memory pressure, the cluster scheduler decides to prioritize local memory for other higher priority VMs, thus by (1) The Cluster Memory Scheduler initiates the process by instructing the Host VMM to redistribute memory capacity of latency tolerant VMs or lower priority VMs toward the CXL memory pool. (2) The VMM then remaps VM's physical memory backend layout, adjusting the proportion between Local(DRAM) and Remote(CXL pooled) memory.(3) Subsequently, color recorder updates the latest host-guest physical memory topology changes to the guest VMs.

At the guest side, (4) the dyno-NUMA adjusts the guest NUMA topology based on the latest host-guest physical memory topology. (5) Throughout this process, the guest temperature profiler continuously monitors applications' memory access patterns. (6) The guest temperature profiler ensures apps' frequently accessed (hot) pages remain in local memory whenever possible through page migration. In the worst case – the faster tier memory capacity is less than the hot memory requirements – the guest will have to sacrifice some hot pages by leaving them in the remote memory tier (noted as the orange bar).

When memory pressure lifted, the system executes these operations in reversely as shown by (1-3) on the right side of the figure, with the VMM redistributing memory from the remote tier back to local DRAM.

5.6 Initial experiment results

Testbed. Currently, the Libra prototype is implemented on a NUMA-based machine. We run experiments on a custom-built server based on the Supermicro X11DPi-N motherboard with a total of 768GB of DRAM and two Intel(R) Xeon(R) Gold 6230R CPUs at 2.1GHz – 26 cores each. To ensure more consistent experimental results, hyperthreading has been disabled. Additionally, all tests are conducted with the CPU performance profile engaged, which includes disabled frequency scaling and turbo boost, to eliminate any variables that could affect the consistency of our results. Libra's effectiveness has been assessed through a series of microbenchmarks conducted on this NUMA setup. We have conducted three experiments designed to highlight the issues present in current tiered memory pooling designs and demonstrate the initial effectiveness of Libra in improving memory utilization in Cloud environments.

5.6.1 Workloads profile

We conducted an experiment using PTE-scan based profiling, with the results illustrated in Fig 5.1. Initially, data was sequentially inserted into an empty Redis-server [150] using the YCSB [176] benchmark. Then, we executed two pairs of YCSB benchmarks that combined Hotspot access patterns (targeting 10% and 20% of the data) and Uniform access patterns. Given that the VM was freshly booted for the first part of the experiment – (A) and (B), profiling the hot memory and deciding on its migration was

relatively straightforward during the hotspot memory access phase. However, after reloading the previously inserted data from the disk, we observed that the memory was sparsely spanning across the entire physical memory address space. As a result in (C) and (D), the performance of the two YCSB Hotspot accesses became similar to that of the uniform access, indicating that profiling for such a workload would require monitoring the entire VM's physical address space.

Fig 5.2 showcases the results from different YCSB memory access patterns, including Zipfian, Latest, and Hotspot. From the observation, it is apparent that the hot memory zones shift over time, particularly because the "Latest access" pattern tends to target the most recently inserted data. This dynamic nature of Cloud workloads, where the Hotspot can shift during runtime, suggests that profiling such workloads and conducting page migration may introduce significant overhead, as the characteristics of the pages are not static over time. This variability can complicate the efficiency of tiered memory management design in current Cloud environments.

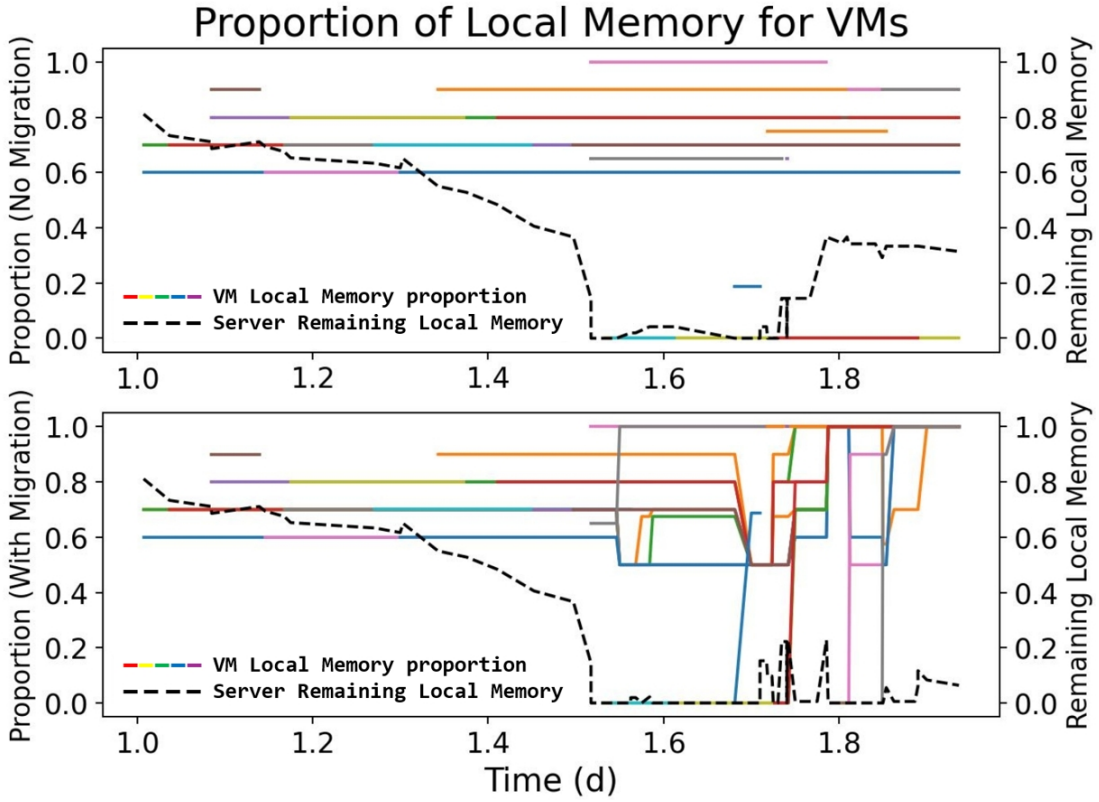


Figure 5.6: *Azure trace simulation, Vanilla vs Libra*,X-axis: the time, normalized to day, start with midnight Each Color Line Segment: Individual VM Lifetime and its Local Memory Proportion point with Y-axis at Left Dash Line: All servers Remaining Local Memory point with Y-axis at Right Upper Fig: Vanilla - VM fixed assigned memory proportion Lower Fig: Libra - VM dynamically assign memory proportion

5.6.2 Cloud simulation

In this experiment, we designed a Python script-based simulator, similar to the one described in [118], to analyze Azure VM traces [54]. Our focus was on a single server

initially loaded with a mix of VMs. Host server memory capacity is normalized to 1 to match the Azure trace data; remote pool memory is simulated with 0.5 memory capacity. To streamline our simulation, we did not include the pooling competition scenario that involves multiple servers. Our simulation demonstrates scenarios where, as the VM arrival pressure increases, we detect memory access and I/O interrupts to assign different priorities to the allocated VMs. This allows us to migrate the memory of lower-priority VMs to increase their proportion of remote memory while reserving the highest proportion of local memory for higher-priority VMs. As the pressure from VM arrivals begins to decline – post peak stage, our simulation showcases how, with the aid of a global scheduler, we can start shifting the VMs back to local memory to optimize local memory usage.

The results of our simulation are presented in Fig 5.6, the top part depicts the vanilla case where VMs are statically allocated with a fixed combination of local and remote memory, while the bottom part shows Libra, which dynamically adjusts the VM memory proportions. First observation: VM arrival pressure in Cloud is change over day. Starting from day 1.6 in the simulation, a decrease in VM arrival pressure leads to more local memory becoming available. Vanilla(Top): VMs allocated with fixed local and remote memory proportions, and local memory is being wasted. However, Libra(bottom) dynamically adjusts VM memory proportions and fully utilizes the local memory. It begins migrating VMs back to local memory. This adaptive management allows Libra to significantly reduce the amount of unused local memory: the server's remaining local memory drops to around 7% on average, compared to the vanilla case where it remains about 36%.

5.6.3 NUMA aware

In this set of experiments, we implemented a simple prototype of Libra. We set up a VM equipped with 10GB of memory, featuring a modified page allocator that receives the current tiered memory boundary from the host via QEMU ivshm. The host adjusts the proportion of memory allocated to the VM, moving 5GB from local to remote memory. We then ran a group of STREAM apps in the guest, with each allocating 1GB of memory.

The results, shown in Fig 5.7, compare the performance between Libra and a Vanilla setup. In the NUMA unaware case: guest is not aware memory topology change; Random Allocation: despite having local memory available in the guest, the memory allocation tends to be random, mixing both local and remote memory. Inefficient: Use remote memory when local memory is available, suboptimal performance. This occurs because the broken NUMA : Guest VM NUMA topology is broken by host page migration. As a result, the guest may allocate memory inefficiently, utilizing remote memory even when local memory is available.

Conversely, with Libra, which is dynamically tiered memory-aware, from the experimental results, we can see that Libra fully utilizes all available local memory before it starts allocating memory from remote. In summary, Libra supports Transparent NUMA Topology: Host updates Guest on the NUMA topology; Tiered Memory Awareness: Fully utilizes all available local memory; Efficient: Guest can choose to allocate memory from either local or remote memory based on different policy.

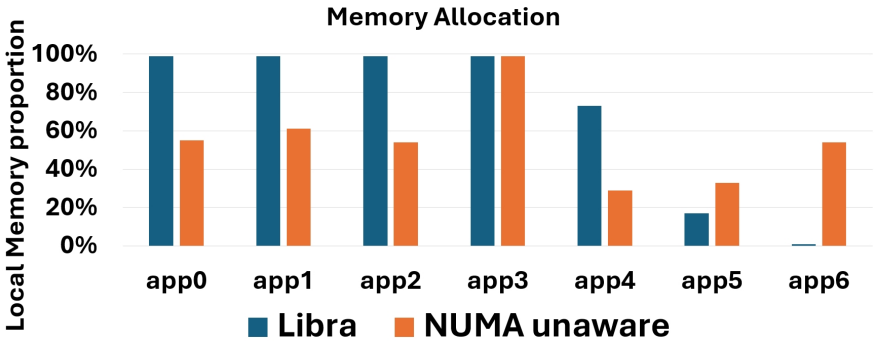


Figure 5.7: *Memory allocation, Libra vs NUMA unaware*, A VM change from 100% local to 70% local memory; Y-axis: Allocated local memory proportion; X-axis: App(STERAM benchmark, memory intensive); Guest allocate 6 apps, record each app memory allocation

5.7 Related Work

Previous works proposed various solutions to improve memory utilization in the Cloud; here we show the history of the research in the Cloud memory utilization topic. **Table 5.1** summarizes previous works, highlighting what they target and what they improved.

5.7.1 Memory ballooning

The most traditional solution to utilize the memory resource for the VM is memory ballooning. Which enables the VM to dynamically resize the memory to adjust the service requirements to maintain an ideal quality of service. Hyperupcall [44] and vProbe [169] mainly target this case. Instead of letting the guest decide when to release the memory, it allows the hypervisor to inject safe eBPF or binary code into the guest for instructing safe and accurate memory reclamation.

Because the hot-added memory can be from anywhere, the memory ballooning could span VM NUMA topology across multiple NUMA nodes. With emerging CXL or NVM-based remote memory, the NUMA spanning problem became inevitable. MHVM [88] and XPV [67] explore the NUMA-spanning problem brought by memory ballooning in real Cloud platforms. MHVM tries to balance the VM placement, which maximizes the likelihood that regular VMs will be fully backed by single NUMA nodes; however, it doesn't consider the case of harvesting the memory from the remote tiered memory. On the other hand, XPV ensures the hot-added memory will be inserted into the correct NUMA node inside the VM. If there is no previous NUMA node, it will create one to handle the newly added memory.

Unfortunately, memory ballooning relies on the memory hot-plug, which requires the VM to provide a continuous range of free physical memory to be hot removed. Which requires the page migration inside the VM; the unmovable page could fail the operation, making this approach inflexible and unable to fast react to dynamically adjustment of the memory requirement change.

5.7.2 Page migration

Another solution is page migration based software tiered memory management. This method involves the host sampling workload memory access, relocating hot data to local memory and cold data to remote memory, thus allowing the workload to dynamically adjust its memory proportions between local and remote storage. Several studies have been conducted to improve the efficiency of both page sampling and migration. Previous works [116, 130, 148, 173, 177] primarily focus on bare-metal scenarios. They explore the use of lightweight sampling tools such as Intel PEBS and have significantly improved page migration by implementing parallelized or asynchronous transparent page migrations, and by accurately detecting partially hot pages within large pages.

Conversely, those works [36, 101, 110, 153, 181] target virtualized environments. Like in the bare-metal case, the host still needs to efficiently sample the guest VM's pages. However, rather than relying on NUMA auto-balancing within the guest, the host manages the migration of hot and cold pages to optimize overall performance. For instance, [36] randomly profiles only 0.5% of a VM's total memory, and [153] uses PML with insights from the VM's page table to reduce the overhead of page sampling while maintaining accuracy. [110] introduces a special NUMA node that exclusively assigns fast memory from local DRAM. The most recent work [181] has proposed a hardware-based tiered memory management design, targeting the Intel flat memory model.

However, these studies generally do not focus on multi-tenant scenarios. While [181] does address the impact of neighboring VMs, its approach is not broadly applicable as it specifically handles cases in the intel flat memory model where only two candidates share the same cache line address and might interfere with each other. This limitation indicates a gap in the current research, highlighting the need for solutions that consider the complexities of multi-tenant environments more comprehensively.

Table 5.1: *Summary of previous works*

Approaches	**Target**	**Contribution(s)**
RAMinate[101]	Virtual & Page migration	initial prototype for VM page migration
Thermostat[36]	Virtual & Page migration	reduce the sampling overhead
HeteroOS[110]	Virtual & Page migration	host-guest co-design
Hyperupcall[110]	Virtual & Ballooning	accurate memory reclaiming
Nimble[177]	Bare-metal & Page migration	reduce page migration cost
XPV[67]	Virtual & Ballooning	NUMA aware virtualization
HeMem[148]	Bare-metal & Page migration	improve sampling efficiency
MHVM[88]	Virtual & Ballooning	dynamical resize VM memory
vProbe[169]	Virtual & Ballooning	DMA safe memory reclaiming
TPP[130]	Bare-metal & Page migration	reduce sampling and migration overhead
Memtis[116]	Bare-metal & Page migration	improve sampling and migration efficiency
Pond[119]	Virtual & Page migration	showcase the CXL-based memory pooing
vTMM[153]	Virtual & Page migration	reduce the sampling overhead
Nomad[173]	Bare-metal & Page migration	asynchronous and transparent page migration
Memstrata[181]	Virtual & Page migration	reduce memory contention in flat memory model

5.8 Conclusion

Page migration based software tiered memory management is inadequate in a virtualization environment: Misplacing pages of a running VM can lead to fragmented physical memory; Variability of workloads increases the challenges for the host to do effective profiling; Cloud multi-tenant scenarios introduce memory contention between different VMs. To address these challenges, we proposed Libra. Libra focuses on VM-level profiling to select VMs that should receive the memory resources, and introduces a memory scheduler that dynamically adjusts the memory proportion allocated to VMs. Our initial prototype results show that Libra can eliminate the randomness of memory allocation and improve memory utilization.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we have thoroughly investigated the redesign of supervisory software to enhance application resource utilization in environments featuring emerging high-speed and shared-memory peripheral interconnects. Specifically, we focused on adapting existing hypervisor designs to better support CPU oversubscription alongside the accelerated I/O capabilities provided by next-generation PCIe interfaces. Additionally, we explored transparent and effective methods for employing CXL-enabled tiered memory systems. Beyond hypervisor enhancements, we examined ways to extend traditional operating systems to operate seamlessly across multiple servers interconnected by CXL-enabled shared memory, including scenarios involving CPUs with diverse ISA. These efforts aim to establish a heterogeneous computing framework for improving resource utilization and performance in modern computing infrastructures by leveraging advanced interconnect technologies like CXL.

The thesis begins with an in-depth introduction to fundamental concepts and emerging technologies, such as the faster PCIe interface and the CXL protocol. It highlights prevalent challenges within current cloud infrastructures, including performance degradation during resource contention, inadequate designs for integrating new hardware, and the lack of support in existing operating systems for heterogeneous computing resources. In summary, this thesis makes the following contributions:

6.1.1 Anubis: Maximizing VMs' IO Performance on Oversubscribing CPUs with Fairness

To address the challenges posed by rapid advancements in interconnects and I/O devices, we developed Anubis, an advanced I/O-aware VM scheduler optimized for Linux KVM — the leading virtual machine monitor in modern cloud computing. Anubis aims to deliver performance close to environments without resource oversubscription, particularly enhancing I/O operations in densely packed virtual environments. Anubis tackles a critical issue that previous research has not fully explored: the inability to consistently guarantee low-latency I/O in virtualized environments due to the CPU resource oversubscription. While earlier studies have primarily focused on direct causes of slow I/O—such as the descheduling of vCPUs handling I/O tasks—Anubis

brings attention to a more subtle yet significant factor: delays in dispatching I/O device interrupts or rescheduling Inter-Processor Interrupts (IPIs) to vCPUs. What sets Anubis apart is its ability to dynamically boost a vCPU's priority when it is engaged in I/O tasks – but only for the duration of those tasks – thereby enhancing responsiveness without requiring any changes to guest software. This precise, context-aware scheduling minimizes unnecessary overhead and maximizes the efficient use of computing resources.

Key features of Anubis include:

1. Responsiveness Enhancement: Anubis enhances vCPU responsiveness by ensuring rapid activation upon the arrival of IO-related interrupts, thereby minimizing delays and reducing the impact of vCPU inactivity.

2. I/O Performance Optimization: During periods of I/O activity, Anubis elevates the priority of the entire I/O vCPU to maximize its operational window, significantly improving the efficiency of I/O processing.

3. Fair Resource Distribution: Anubis implements a debt system to balance resource distribution across VMs, maintaining fairness while optimizing I/O performance.

Overall, Anubis introduces a nuanced approach to VM scheduling that meets the dynamic demands of modern cloud infrastructures, ensuring high performance, responsiveness, and equitable resource allocation.

6.1.2 Stramash: A Fused-kernel Operating System For Cache Coherent, Heterogeneous-ISA Platforms

Advancements in inter-connect technology, CXL, now allow shared memory access across different machines. Inspired by these developments, we developed Stramash, a groundbreaking operating system kernel design that enables a unified OS to function across multiple machines, potentially with diverse ISAs. Given the potential benefits highlighted by academic research on heterogeneous-ISA cache-coherent platforms and the development of new cache coherence interconnects for connecting heterogeneous-ISA processors on the same platform, it is crucial to explore the optimal architecture for operating system software on such platforms.

In this work, we propose the fused-kernel, a multi-kernel operating system architecture that fully leverages cache-coherent shared memory to facilitate communication between kernels. We implemented this new design in a prototype based on the Linux kernel, supporting both ARM and x86 ISAs. This demonstrates that the fused-kernel OS concept can be integrated into traditional monolithic operating systems.

Due to the absence of commercially available hardware for our target platforms, we constructed a hardware simulator named Stramash-QEMU, which extends QEMU with a cache plugin for precise memory simulation. Using Stramash-QEMU, we evaluated the performance of our fused-kernel OS design against Popcorn Linux, an existing multi-kernel OS designed for heterogeneous hardware. Our findings indicate that Stramash significantly outperforms Popcorn Linux in terms of performance, showcasing the advantages of our innovative fused-kernel approach.

6.1.3 Libra: The Art of Memory Placement in Virtualization

The evolution of tiered memory technology has led us to develop Libra, an innovative extension of the hypervisor memory subsystem specifically tailored to enhance tiered memory management within cloud environments. Libra is designed to dynamically adjust memory allocation between local and extended memory sources, thereby optimizing overall memory utilization for VMs.

Traditional software-based tiered memory management techniques, such as page migration, often fall short in virtualized settings due to several key challenges:

1. Memory Fragmentation: Misplacing pages of a running VM can result in fragmented physical memory, degrading system performance.

2. Workload Variability: The diverse and unpredictable nature of cloud workloads complicates the host's ability to effectively profile and manage memory.

3. Multi-Tenant Memory Contention: In cloud environments where multiple tenants share resources, inherent competition for memory can lead to contention and reduced efficiency.

To overcome these challenges, we designed Libra with a focus on VM-level profiling. Unlike traditional methods that manage memory at a granular page level, Libra identifies and prioritizes VMs based on their memory needs and operational characteristics. It incorporates a memory scheduler that dynamically adjusts the proportion of memory allocated to each VM, ensuring resources are distributed based on current demand and usage patterns.

Preliminary results from our prototype implementation of Libra demonstrate its ability to eliminate the randomness traditionally associated with memory allocation in cloud environments. By providing a more structured and responsive approach to memory management, Libra significantly improves memory utilization, making it a promising solution for modern cloud platforms facing the complexities of tiered memory management.

6.2 Future works

As cloud connectivity continues to evolve, the potential for resource sharing among multiple virtual machines (VMs) becomes increasingly feasible. With workloads becoming more segmented into function-as-a-service (FaaS) structures, VMs remain essential as the foundational environment for hosting these containers.

Anubis. Our observations from Anubis reveal that the current default operating system scheduler, the Completely Fair Scheduler (CFS), is primarily designed to distribute CPU time equitably among different workloads or to adjust priority by modifying the rate of virtual runtime accumulation. This allows higher-priority tasks to consume more CPU time. However, managing complex workloads that require comprehensive software and hardware stacks necessitates more than just balancing CPU time; it demands a holistic scheduling approach. Inappropriate scheduling decisions can exacerbate performance interference among co-running workloads. Consequently, there is a clear need for more detailed profiling to enable more precise scheduling decisions. Minimizing the frequency of context switches is also crucial to enhance performance

and efficiency. Future work should focus on developing a resource contention-aware scheduler that is closely integrated with the hypervisor to address these challenges. This co-design approach would allow for more adaptive and intelligent resource management, catering to the intricate dynamics of modern cloud environments.

Stramash. In today's landscape, applications are demanding increasingly large memory capacities, especially with the rise of in-memory databases and large AI models. With CXL, data center managers can leverage tools like Direct Access (DAX) or memory hot-plugging to create vast shared memory pools. These pools can span multiple nodes and CXL devices—from a server's main memory to extended memory modules and even the private memory of CXL-connected devices—all interconnected through the peripheral bus. This significantly increases the total memory capacity available to a single machine. However, while shared memory on a single machine is achievable, scaling this capability across multiple nodes presents challenges as the shared memory pool needs to be accessed by several nodes at the same time. Current operating systems do not yet enable seamless sharing; instead, they reassign memory capacities among different machines. Consequently, even if memory is shared across machines, users must run multiple processes—one per node—connecting to the same shared memory. This limitation raises a critical question: What does shared memory across machines actually mean?

Potential solutions, such as concepts like /dev/rshm, could address this issue, but the operating system must evolve to support true shared memory between nodes. Moreover, when an application requires more CPUs to work on the same piece of data, today's architectures often revert to distributed models. Why can't the operating system provide more support in these scenarios? Why must developers shift to distributed programming to process shared data when additional CPUs are needed? These challenges highlight a critical demand in future data centers: enabling efficient memory sharing across servers. While DAX provides a basic mechanism, it lacks control over data management. To effectively share hardware resources across multiple applications running on different nodes, a more sophisticated solution is required. This is where the operating system must evolve. Current OS designs are not equipped to manage this level of resource sharing, underscoring the need for a new OS architecture. One might consider the multi-kernel approach as a potential solution for these futuristic platforms. However, prior research, such as the Barrelfish operating system[61], has shown scalability issues for cache-coherent shared memory even on a small number of cores. As the number of cores and the complexity of the interconnect grows, hardware cache-coherence protocols become increasingly expensive. Does CXL change this assumption? When we talk about scaling out instead of scaling up, will this limitation still hold?

We need to explore the benefits that shared memory can provide, for example, an essential large-scale application case: in-memory database clusters. Today, various in-memory database solutions—such as those from Oracle [12] – operate across clustered nodes. However, despite their reliance on shared data, these systems still adopt a distributed programming model. In these environments, the shared data consistency and its duplication still be entirely managed at the user application level. This is primarily because the current OS lacks the capability to handle cross-node memory sharing.

Therefore, can the OS leverage CXL-based shared memory to enhance data sharing across different nodes? Can this CXL-based shared memory overcome scalability limitations, or is there a trade-off to explore in collaboration with message passing? Will the scalability limitations of shared memory become a critical issue when scaling out? Which kernel subsystems can be shared like in SMP, and which should remain independent and act individually? Can we adopt a combination of regular kernels and lightweight kernels, similar to the big.LITTLE architecture?

These concepts aren't new, but the landscape of our research has evolved significantly, urging us to innovate upon existing solutions. We might find ourselves revisiting the principles of SMP, but this time envisioning a server not as a collection of discrete computing cores, but as a unified, general computing entity.

Libra. Libra addresses key shortcomings in current operating system designs, particularly in how they integrate with emerging hardware technologies like memory machines[20, 102]. With the advent of new extended memory devices, it is evident that a robust "control plane" is necessary to manage memory allocation effectively. This control should transcend the OS level, shifting responsibility to a global scheduler capable of managing memory distribution across different VMs, rather than merely handling it at the page level, which predominantly focuses on applications. Current methods often overlook the complexities of profiling at the page level, which can be intricate and demanding due to the diverse and dynamic nature of VM workloads. To address this, a more sophisticated approach is required that enhances memory management at the VM level. This involves developing a framework that not only allocates memory more efficiently but also dynamically adjusts to the changing needs of VMs based on real-time usage data.

Designing such a system would require extensive development in several key areas: **Dynamic Memory Allocation:** Implementing algorithms that can adjust memory allocations in real time based on current demand and workload characteristics. **VM-Level Profiling:** Developing tools that can accurately profile VMs for better resource allocation, moving beyond traditional application-centric approaches. **Integration with Hardware:** Ensuring that the memory management system can seamlessly integrate with and fully leverage the capabilities of new memory hardware technologies. **Resource Balancing:** Creating mechanisms that ensure fair memory distribution among VMs to prevent any single VM from monopolizing resources, thereby maintaining balanced performance across the cloud environment. Enhancing Libra to incorporate these elements would make it a more powerful tool for managing memory resources in cloud computing environments, ensuring it keeps pace with evolving hardware innovations and the increasing complexity of cloud-based architectures.

Appendix A

Works and Publications

During the course of the PhD, I worked on several topics, including what was presented in this Thesis. Nine publications originated from my PhD work, seven of which are already published (Asplos'25, HCDS'25, Middleware'24, SoCC'23, EuroSys'23, APSys'22, TOCS), and two are under submission (HotOS'25) or for future submission (Libra). Of the 9 publications, I am the first author on 7, co-first author on 1 and co-author on 1 of those. Before my PhD, I was a co-author of a publication accepted at VEE'20. A list of all my publications follows.

A.1 Works under submission

- **T. Xing**, A. Barbalace, "CXL shared memory: SMP or distributed OS? Inter-kernel IPC and namespaces!" *Under summitsion of **HotOS'25***

- "Libra: The Art of Tiered Memory Virtual Machines Placement" *For future Submission*

A.2 Published works during the PhD

- **T. Xing**, C. Xiong, T. Wei, J. Balkind, B. Ravindran, A. Sanchez, A. Barbalace, "Stramash: A Fused Kernel Operating System For Cache-Coherent, Heterogeneous-ISA Systems," *in Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems(**Asplos'25**), Rotterdam, 2025*

- **T. Xing**, A. Barbalace *"Rethinking Applications' Address Space with CXL shared memory pools," in Proceedings of 4th Workshop on Heterogeneous Composable and Disaggregated Systems(**HCDS'25**), Rotterdam, 2025*

- D. Mvondo, **T. Xing**, A. Barbalace, "UTwinVM: Reliable hints on the effects of hypervisor updates on VMs in the Cloud," *in Proceedings of the 25th ACM/IFIP International Middleware Conference(**Middleware'24**)*, Hong Kong, Pages 103 - 116, 2024

- **T. Xing**, C. Xiong, J. Picorel, Y. Chuan, A. Barbalace "Maximizing VMs' IO Performance on Overcommitted CPUs with Fairness", *in Proceedings of the 2023 ACM Symposium on Cloud Computing(**SoCC'23**), Santa Cruz, Pages 93–108, 2023*

- H. Chuang, K. Manaouil, **T. Xing**, A. Barbalace, P. Olivier, B. Heerekar, B. Ravindran, "Aggregate VM: Why Reduce or Evict VM's Resources When You Can Borrow Them From Other Nodes," **(Co-first author),** *in Proceedings of the Eighteenth European Conference on Computer Systems* (**EuroSys '23**), Rome, Pages 469–487, 2023

- **T. Xing**, H. Tajbakhsh, I. Haque, M. Honda, A. Barbalace, "Towards Portable End-to-End Network Performance Characterization of SmartNICs," *in Proceedings of 13th ACM Asia-Pacific Workshop on Systems **(APSys'22)**, Online, Pages 46–52, 2022*

- **T. Xing**, A. Barbalace, P. Olivier, ML. Karaoui, W. Wang, B. Ravindran, "H-Container: Enabling Heterogeneous-ISA Container Migration in Edge Computing," *ACM Transactions on Computer Systems **(TOCS)**, Volume 39, Issue 1-4, 2022*

A.3 Published works before the PhD

- A. Barbalace, ML. Karaoui, W. Wang, **T. Xing**, P. Olivier, B. Ravindran, "Edge computing: the case for heterogeneous-ISA container migration", *in Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments **(VEE '20)***,Pages 73–87, Online, 2020

Bibliography

[1] Amazon ec2. https://aws.amazon.com/cn/ec2/.

[2] Amd infinity fabric™ link. https://www.amd.com/content/dam/amd/en/documents/instinct-tech-docs/other/56978.pdf.

[3] Amd unveils their embedded+ architecture, ryzen embedded with versal together. https://www.anandtech.com/show/21254/amd-unveils-their-embedded-architecture-ryzen-embedded-with-ve

[4] Architecting chips for high-performance computing. https://semiengineering.com/architecting-chips-for-high-performance-computing/.

[5] Arm cortex-a72 execution and load/store. http://sandsoftwaresound.net/arm-cortex-a72-execution-and-load-store/.

[6] Assessing cavium's thunderx2: The arm server dream realized at last. https://monimega.com/blog/2018/05/23/assessing-caviums-thunderx2-the-arm-server-dream-realized-at-l

[7] Bluefile. https://www.nvidia.com/en-us/networking/products/data-processing-unit/.

[8] Cache modelling tcg plugin. https://www.qemu.org/2021/08/19/tcg-cache-modelling-plugin/.

[9] Cascade lake - microarchitectures - intel. https://en.wikichip.org/wiki/intel/microarchitectures/cascade_lake.

[10] Characterizing shared-memory applications: A case study of the nas parallel benchmarks. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=3c26c62a6258f03ea22e441f579bbdbfa2f05c2c.

[11] Compiler support for application migration in heterogeneous-isa platforms. https://eurosys2015.labri.fr/posters/p46.pdf.

[12] Database in-memory. https://www.oracle.com/uk/database/in-memory/.

[13] Dual-core artificial intelligent processor sg200x. https://en.sophgo.com/sophon-u/product/introduce/sg200x.html.

[14] Ibm vm 50th anniversary. https://www.vm.ibm.com/history/50th/index.html.

[15] Intel broadwell. https://www.7-cpu.com/cpu/Broadwell.html.

[16] Introduction to large system extensions. https://learn.arm.com/learning-paths/servers-and-cloud-computing/lse/intro/.

[17] Isa. https://www.computerlanguage.com/results.php?definition=VESA%2FISA.

[18] Memory balloon driver. https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-5B45CEFA-6CC6-49F4-A3C7-776AAA22C2A2.html.

[19] Nas parallel benchmarks, rnr-94-007 (pdf-425kb) for is, ep, cg, mg, ft, bt, sp, lu. https://www.nas.nasa.gov/assets/nas/pdf/techreports/1994/rnr-94-007.pdf.

[20] New memory sharing model. https://memverge.com/memory-machine-cxl-fabric-attached-memory/.

[21] The next platform. cxl and gen-z iron out a coherent interconnect strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-acoherent-interconnect-strategy/.

[22] An openinfra foundation project. https://www.openstack.org/.

[23] Pci. https://www.webopedia.com/definitions/pci/.

[24] Pcie. https://www.computerlanguage.com/results.php?definition=PCIE.

[25] Pcie6.0 specification. https://pcisig.com/pci-express-6.0-specification.

[26] Popcorn-compiler. https://github.com/ssrg-vt/popcorn-compiler.

[27] Popcorn linux: A compiler and runtime for state transformation between heterogeneous-isa architectures. https://www.ssrg.ece.vt.edu/theses/PhdProposal_Lyerly.pdf.

[28] Ruby:mesi three level. https://www.gem5.org/documentation/general_docs/ruby/.

[29] Samsung smartssd. https://semiconductor.samsung.com/ssd/smart-ssd/.

[30] Scaling the facebook data warehouse to 300 pb. https://engineering.fb.com/2014/04/10/core-infra/scaling-the-facebook-data-warehouse-to-300-pb/.

[31] ssrg-vt/popcorn-kernel. https://github.com/ssrg-vt/popcorn-kernel/blob/main/include/popcorn/pcn_kmsg.h#L88.

[32] Upmem. https://www.upmem.com/.

[33] Vesa. https://www.computerlanguage.com/results.php?definition=VESA+local+bus.

[34] vsphere. https://www.vmware.com/products/cloud-infrastructure/vsphere.

[35] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 419–434, USA, 2020. USENIX Association.

[36] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 631–644, New York, NY, USA, 2017. Association for Computing Machinery.

[37] Gustavo Alonso, Timothy Roscoe, David Cock, Mohsen Ewaida, Kaan Kara, Dario Korolija, David Sidler, and Zeke Wang. Tackling hardware/software co-design from a database perspective. In *Conference on Innovative Data Systems Research*, CIDR 2020, 2020.

[38] Amazon. EC2 Instances (A1) Powered by Arm-Based AWS Graviton Processors, 2018. https://aws.amazon.com/blogs/aws/new-ec2-instances-a1-powered-by-arm-based-aws-graviton-process

[39] Amazon. AWS Lambda Website, 2020. https://aws.amazon.com/lambda.

[40] Amazon. How Amazon ECS manages CPU and memory resources, 2022. https://aws.amazon.com/blogs/containers/how-amazon-ecs-manages-cpu-and-memory-resources/.

[41] Amazon. Burstable performance instances, 2023. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-performance-instances.html.

[42] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting

VMs in cloud platforms. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751. USENIX Association, November 2020.

[43] AMD. Amd research instruction based sampling toolkit. https://github.com/jlgreathouse/AMD_IBS_Toolkit, 2024.

[44] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 97–112, Boston, MA, July 2018. USENIX Association.

[45] Apache. ab - apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, 2023.

[46] Giuseppe Attardi, A Baldi, U Boni, F Carignani, G Cozzi, A Pelligrini, E Durocher, I Filotti, Wang Qing, M Hunter, et al. Techniques for dynamic software migration. In *Proceedings of the 5th Annual ESPRIT Conference (ESPRIT'88)*, volume 1, 1988.

[47] AWS. Bbc delivers live, uhd coverage of uefa euros and wimbledon with aws, 2023. https://aws.amazon.com/cn/blogs/media/bbc-delivers-live-uhd-coverage-of-uefa-euros-and-wimbledon-with-aws/

[48] AWS. Explore.org live streams nature cams to global audiences with aws, 2023. https://aws.amazon.com/cn/blogs/media/explore-org-live-streams-nature-cams-to-global-audiences-with-aws/.

[49] AWS. Lamp server on aws, 2023. https://aws.amazon.com/marketplace/pp/prodview-gqnnpbafrkkys.

[50] AWS. Partner success with aws, 2023. https://aws.amazon.com/partners/success/.

[51] AWS. Washington post's arc publishing platform uses aws to transform the broadcast landscape, 2023. https://aws.amazon.com/cn/blogs/media/washington-posts-arc-publishing-platform-uses-aws-to-transform-the-b

[52] AWS, 2024. https://aws.amazon.com/cn/ec2/.

[53] Azure, 2024. https://learn.microsoft.com/en-us/azure/virtual-machines/b-series-cpu-credit-model/b-series-cpu-credit-model.

[54] Azure. Azurepublicdataset. https://github.com/Azure/AzurePublicDataset, 2024.

[55] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M. Nguyen, Yaosheng Fu, Florian Zaruba, and et al. BYOC: A "bring your own core" framework for heterogeneous-ISA research. In

Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, page 699–714, New York, NY, USA, 2020. Association for Computing Machinery.

[56] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. Openpiton: An open source manycore research framework. In *ACM SIGARCH Computer Architecture News*, volume 44, pages 217–232. ACM, 2016.

[57] Antonio Barbalace, Mohamed L. Karaoui, Wei Wang, Tong Xing, Pierre Olivier, and Binoy Ravindran. Edge computing: The case for heterogeneous-isa container migration. VEE '20, 2020.

[58] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-isa datacenters. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 645–659, New York, NY, USA, 2017. ACM.

[59] 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, pages 29:1–29:16, 2015.

[60] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, CCS '03, pages 281–289, New York, NY, USA, 2003. ACM.

[61] 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, pages 29–44, 2009.

[62] 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, page 29–44, New York, NY, USA, 2009. Association for Computing Machinery.

[63] Sharath K. Bhat, Ajithchandra Saya, Hemedra K. Rawat, Antonio Barbalace, and Binoy Ravindran. Harnessing energy efficiency of heterogeneous-isa platforms. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '15, pages 6–10, New York, NY, USA, 2015. ACM.

[64] Blueprint, 2022. https://blueprints.launchpad.net/nova/+spec/nova-change-default-overcommit-values.

[65] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. The battle of the schedulers: Freebsd ule vs. linux cfs. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 85–96, 2018.

[66] Broadcom. *Stingray PS225*, 2018 (accessed June 30, 2020).

[67] Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. When extended para - virtualization (xpv) meets numa. EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Kevin Burns, Antonio Barbalace, Vincent Legout, and Binoy Ravindran. Kairosvm: Deterministic introspection for real-time virtual machine hierarchical scheduling. In *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pages 1–8, 2014.

[69] Kevin Burns, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. Prvm: A multicore real-time virtualization scheduling framework with probabilistic timing guarantees. *SIGBED Rev.*, 16(3):14–20, nov 2019.

[70] CCIX Consortium. Cache Coherent Interconnect for Accelerators (CCIX). http://www.ccixconsortium.com/, 2017.

[71] Luwei Cheng and Cho-Li Wang. Vbalance: Using interrupt load balance to improve i/o performance for smp virtual machines. SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[72] Shenghsun Cho, Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. Flick: Fast and lightweight isa-crossing call for heterogeneous-isa environments. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 187–198, 2020.

[73] Huawei Cloud. Elastic cloud server (ecs), 2023. https://www.huaweicloud.com/intl/en-us/product/ecs.html.

[74] Huawei Cloud. A summary list of x86 ecs specifications, 2023. https://support.huaweicloud.com/intl/en-us/productdesc-ecs/ecs_01_0014.html.

[75] Compute Express Link Consortium, Inc. *Compute Express Link (CXL) Specification*, 3.0 edition, 2022. Available: Compute Express Link Consortium, https://www.computeexpresslink.org/download-the-specification.

[76] Key concepts and definitions for burstable performance instances. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/burstable-credits-baseline-concepts.html, 2023.

[77] Emilio G. Cota, Paolo Bonzini, Alex Bennée, and Luca P. Carloni. Cross-isa machine emulation for multicores. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 210–220. IEEE Press, 2017.

[78] CXL Consortium. Cxl specification, 2024. https://www.computeexpresslink.org/download-the-specification.

[79] Mehiar Dabbagh, Bechir Hamdaoui, Mohsen Guizani, and Ammar Rayes. Efficient datacenter resource utilization through cloud resource overcommitment. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 330–335, 2015.

[80] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272, 2012.

[81] Matthew DeVuyst, Ashish Venkat, and Dean M. Tullsen. Execution migration in a heterogeneous-isa chip multiprocessor. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 261–272, New York, NY, USA, 2012. ACM.

[82] Xiaoning Ding, Phillip B. Gibbons, and Michael A. Kozuch. A hidden cost of virtualization when scaling multicore applications. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 13)*, San Jose, CA, June 2013. USENIX Association.

[83] Xiaoning Ding, Phillip B. Gibbons, Michael A. Kozuch, and Jianchen Shan. Gleaner: Mitigating the Blocked-Waiter wakeup problem for virtualized multicore applications. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 73–84, Philadelphia, PA, June 2014. USENIX Association.

[84] Michael Drescher, Vincent Legout, Antonio Barbalace, and Binoy Ravindran. A flattened hierarchical scheduler for real-time virtualization. In *Proceedings of the 13th International Conference on Embedded Software*, EMSOFT '16. Association for Computing Machinery, 2016.

[85] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Christopher Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023,

page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[86] Yaosheng Fu and David Wentzlaff. Prime: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, 2014.

[87] Yaosheng Fu and David Wentzlaff. Prime: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–125, 2014.

[88] Alexander Fuerst, Stanko Novaković, Íñigo Goiri, Gohar Irfan Chaudhry, Prateek Sharma, Kapil Arya, Kevin Broas, Eugene Bak, Mehmet Iyigun, and Ricardo Bianchini. Memory-harvesting vms in cloud platforms. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 583–594, New York, NY, USA, 2022. Association for Computing Machinery.

[89] Gal Beniamini, Project Zero. Over the air: Exploiting broadcom's wi-fi stack (part 1). `https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html`.

[90] Sahan Gamage, Cong Xu, Ramana Rao Kompella, and Dongyan Xu. Vpipe: Piped i/o offloading for efficient data movement in virtualized clouds. SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[91] Joachim Gehweiler and Michael Thies. Thread migration and checkpointing in java. *Heinz Nixdorf Institute, Tech. Rep. tr-ri-10*, 315, 2010.

[92] Google. Google Cloud Functions, 2020. `https://cloud.google.com/functions`.

[93] Google. Get more from every core: Announcing cpu overcommit for compute engine, 2022. `https://cloud.google.com/blog/products/compute/cpu-overcommit-for-sole-tenant-nodes-now-ga`.

[94] Google Cloud. Overcommit cpus on sole-tenant vms, 2024. `https://cloud.google.com/compute/docs/nodes/overcommitting-cpus-sole-tenant-vms`.

[95] Ori Hadary, Luke Marshall, Ishai Menache, Abhisek Pan, Esaias E Greeff, David Dion, Star Dorminey, Shailesh Joshi, Yang Chen, Mark Russinovich, and Thomas Moscibroda. Protean: VM allocation service at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 845–861. USENIX Association, November 2020.

[96] Hadoop, 2023. `https://hadoop.apache.org/`.

[97] Red Hat, 2017. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/6.0_technical_notes/deployment.

[98] HBase, 2023. https://hbase.apache.org/.

[99] Hadoop Distributed File System (HDFS™). https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesi, 2023.

[100] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[101] Takahiro Hirofuchi and Ryousei Takano. Raminate: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, page 112–125, New York, NY, USA, 2016. Association for Computing Machinery.

[102] Hokyoon Lee and Jungmin Choi (SK hynix). Cxl-based memory disaggregation for hpc and ai workloads, 2024. https://sc23.supercomputing.org/proceedings/exhibitor_forum/exhibitor_forum_pages/exforum119.html.

[103] IBM, 2022. https://www.ibm.com/docs/en/cic/1.1.3?topic=SSLL2F_1.1.3/com.ibm.cloudin.doc/admintasks/configuring/customizing/allocation_ratio_templates.htm.

[104] Intel. An introduction to the intel quickpath interconnect, 2009.

[105] iperf3. https://github.com/esnet/iperf, 2023.

[106] Kenta Ishiguro, Naoki Yasuno, Pierre-Louis Aublin, and Kenji Kono. Mitigating excessive vcpu spinning in vm-agnostic kvm. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE 2021, page 139–152, New York, NY, USA, 2021. Association for Computing Machinery.

[107] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis C. M. Lau, Yuexuan Wang, and Yuangang Wang. Effectively mitigating i/o inactivity in vcpu scheduling. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 267–279, USA, 2018. USENIX Association.

[108] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. *SIGARCH Comput. Archit. News*, 43(3S):158–169, June 2015.

[109] Ardalan Kangarlou, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. vsnoop: Improving tcp throughput in virtualized environments via acknowledgement offload. In *SC '10: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–11, 2010.

[110] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. ISCA '17, page 521–534, New York, NY, USA, 2017. Association for Computing Machinery.

[111] J. Kay and P. Lauder. A fair share scheduler. *Commun. ACM*, 31(1):44–55, jan 1988.

[112] Linux kernel, 2023. `https://elixir.bootlin.com/linux/v4.14.325/source/arch/x86/kernel/apic/apic.c`.

[113] Linux kernel, 2023. `https://elixir.bootlin.com/linux/v4.14.325/source/arch/x86/kernel/apic/apic_flat_64.c`.

[114] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joonwon Lee. Task-aware virtual machine scheduling for i/o performance. VEE '09, page 101–110, New York, NY, USA, 2009. Association for Computing Machinery.

[115] Linux KVM, 2023. `https://www.linux-kvm.org/page/Main_Page`.

[116] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. Memtis: Efficient memory tiering with dynamic page classification and page size determination. SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[117] LEMP. `https://lemp.io/`, 2023.

[118] Philip Levis, Kun Lin, and Amy Tai. A case against cxl memory pooling. HotNets '23, page 18–24, New York, NY, USA, 2023. Association for Computing Machinery.

[119] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

[120] Katie Lim, Jonathan Balkind, and David Wentzlaff. JuxtaPiton: Enabling heterogeneous-isa research with RISC-V and SPARC FPGA soft-cores. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 184–184, New York, NY, USA, 2019. ACM.

[121] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 285–300, 2014.

[122] Felix Xiaozhu Lin, Zhen Wang, and Lin Zhong. K2: A mobile operating system for heterogeneous coherence domains. *ACM Trans. Comput. Syst.*, 33(2):4:1–4:27, June 2015.

[123] Scott D. Lowe. *Best Practices for Oversubscription of CPU, Memory and Storage in vSphere Virtual Environments*. Dell.

[124] Hui Lu, Brendan Saltaformaggio, Ramana Kompella, and Dongyan Xu. Vfair: Latency-aware fair storage scheduling via per-io cost-based differentiation. SoCC '15, page 125–138, New York, NY, USA, 2015. Association for Computing Machinery.

[125] Hui Lu, Cong Xu, Cheng Cheng, Ramana Kompella, and Dongyan Xu. vhaul: Towards optimal scheduling of live multi-vm migration for multi-tier applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 453–460, 2015.

[126] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[127] Daniel Lustig, Caroline Trippel, Michael Pellauer, and Margaret Martonosi. Armor: Defending against memory consistency model mismatches in heterogeneous architectures. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 388–400, 2015.

[128] LWN, 2011. `https://lwn.net/Articles/444503/`.

[129] Robert Lyerly, Antonio Barbalace, Christopher Jelesnianski, Vincent Legout, Anthony Carno, and Binoy Ravindran. Operating system process and thread migration in heterogeneous platforms.

[130] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[131] Mellanox Technologies. Bluefield multicore system on chip, 2017. `http://www.mellanox.com/related-docs/`

npu-multicore-processors/PB_Bluefield_SoC.pdf. Online, accessed 01/05/2019.

[132] Microsoft. Microsoft Azure Functions, 2020. https://azure.microsoft.com/en-us/services/functions.

[133] Microsoft, 2023. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes-b-series-burstable-workload-example.

[134] MongoDB. https://www.mongodb.com, 2023.

[135] NASA Advanced Supercomputing Division. NAS parallel benchmarks. https://tinyurl.com/y47k95cc.

[136] Ricardo Neri, 2022. https://www.spinics.net/lists/kernel/msg4348466.html.

[137] Nginx. https://nginx.org/, 2023.

[138] Edmund B Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. Helios: heterogeneous multiprocessing with satellite kernels. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 221–234. ACM, 2009.

[139] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling i/o in virtual machine monitors. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, page 1–10, New York, NY, USA, 2008. Association for Computing Machinery.

[140] OpenCAPI Consortium. Welcome to OpenCAPI Consortium. http://opencapi.org/, 2017.

[141] OpenEuler, 2022. https://docs.openeuler.org/en/docs/20.03_LTS_SP2/docs/Virtualization/best-practices.html.

[142] Openstack, 2022. https://docs.openstack.org/arch-design/design-compute/design-compute-overcommit.html.

[143] Oracle, 2023. https://www.oracle.com/uk/a/ocom/docs/why-kvm-is-winning.pdf.

[144] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. Profiling dynamic data access patterns with controlled overhead and quality. In *Proceedings of the 20th International Middleware Conference Industrial Track*, Middleware '19, page 1–7, New York, NY, USA, 2019. Association for Computing Machinery.

[145] Mongodb performance test. https://github.com/idealo/mongodb-performance-test, 2023.

[146] PostMark. https://www.filesystems.org/docs/auto-pilot/Postmark.html, 2023.

[147] Xen Project, 2023. https://xenproject.org/.

[148] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[149] Redis. https://redis.io/, 2023.

[150] Redis-benchmark. https://redis.io/docs/management/optimization/benchmarks/, 2023.

[151] Phil Rogers. Heterogeneous system architecture overview. In *Hot Chips*, volume 25, 2013.

[152] R.R. Schaller. Moore's law: past, present and future. *IEEE Spectrum*, 34(6):52–59, 1997.

[153] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. Vtmm: Tiered memory management for virtual machines. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 283–297, New York, NY, USA, 2023. Association for Computing Machinery.

[154] Mohammad Shahrad, Rodrigo Fonseca, Íñigo 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 *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC'20, USA, 2020. USENIX Association.

[155] 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)*, pages 205–218. USENIX Association, July 2020.

[156] Jianchen Shan, Weiwei Jia, and Xiaoning Ding. Rethinking multicore application scalability on big virtual machines. In *2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)*, pages 694–701, 2017.

[157] D. Sharma. Compute express link (cxl): Enabling heterogeneous data-centric computing with heterogeneous memory hierarchy. *IEEE Micro*, 43(02):99–109, mar 2023.

[158] K. Sinha, V. P. Kemerlis, and S. Sethumadhavan. Reviving instruction set randomization. In *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 21–28, May 2017.

[159] Peter Smith and Norman C Hutchinson. Heterogeneous process migration: The tui system. *Software: Practice and Experience*, 28(6):611–639, 1998.

[160] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis C. M. Lau. Preserving i/o prioritization in virtualized oses. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 269–281, New York, NY, USA, 2017. Association for Computing Machinery.

[161] sysbench. `https://github.com/akopytov/sysbench`, 2023.

[162] Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[163] Texas Instruments. OMAP4430 Multimedia Device Silicon Revision 2.x Version AP Technical Reference Manual, 2014. `https://www.ti.com/lit/pdf/swpu231?keyMatch=OMAP4430`.

[164] Twitch, 2023. `https://blog.twitch.tv/en/2017/10/10/live-video-transmuxing-transcoding-f-fmpeg-vs-twitch-transcoder-part`

[165] Ashish Venkat, Sriskanda Shamasunder, Hovav Shacham, and Dean M. Tullsen. Hipstr: Heterogeneous-isa program state relocation. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 727–741, New York, NY, USA, 2016. ACM.

[166] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: design of a heterogeneous-isa chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 121–132. IEEE Press, 2014.

[167] Ashish Venkat and Dean M. Tullsen. Harnessing isa diversity: Design of a heterogeneous-isa chip multiprocessor. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 121–132, Piscataway, NJ, USA, 2014. IEEE Press.

[168] Vmware, 2022. `https://www.reddit.com/r/vmware/comments/dl2bt8/do_you_overcommit_cpu_in_your_environement/`.

[169] Yaohui Wang, Ben Luo, and Yibin Shen. Efficient memory overcommitment for I/O passthrough enabled VMs via fine-grained page meta-data management. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 769–783, Boston, MA, July 2023. USENIX Association.

[170] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. Smartharvest: harvesting idle cpus safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer*

Systems, EuroSys '21, page 1–16, New York, NY, USA, 2021. Association for Computing Machinery.

[171] Intel® 64 Architecture x2APIC Specification. https://www.naic.edu/~phil/software/intel/318148.pdf, 2023.

[172] xen, 2013. https://wiki.xenproject.org/wiki/Credit2_Scheduler.

[173] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. Nomad: Non-Exclusive memory tiering via transactional page migration. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 19–35, Santa Clara, CA, July 2024. USENIX Association.

[174] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Rao Kompella, and Dongyan Xu. Vslicer: Latency-aware virtual machine scheduling via differentiated-frequency cpu slicing. HPDC '12, page 3–14, New York, NY, USA, 2012. Association for Computing Machinery.

[175] Cong Xu, Brendan Saltaformaggio, Sahan Gamage, Ramana Rao Kompella, and Dongyan Xu. Vread: Efficient data access for hadoop in virtualized clouds. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, page 125–136, New York, NY, USA, 2015. Association for Computing Machinery.

[176] Yahoo. Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB, 2023.

[177] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

[178] Al Yanes. Doubling bandwidth in under two years: Pci express® base specification revision 5.0, version 0.9 is now available to members, 2018. https://bit.ly/2ClJ9AT, Online, accessed 01/05/2019.

[179] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *OSDI*, volume 14, pages 17–31, 2014.

[180] Olive Zhao, 2021. https://forum.huawei.com/enterprise/en/why-are-huawei-cloud-computing-products-switched-from-xen-to-k thread/818617-893.

[181] Yuhong Zhong, Daniel S. Berger, Carl Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with CXL in virtualized environments. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 37–56, Santa Clara, CA, July 2024. USENIX Association.